UiO : Department of Informatics
University of Oslo

SIREN - Scalable Infrastructure for laRge scalE vm communicatioN

Kjetil Skansen Elfving

Master's Thesis Spring 2014

SIREN - Scalable Infrastructure for laRge scalE vm communicatioN

Kjetil Skansen Elfving

20th May 2014

Abstract

This thesis aims to explore the possibilities of communicating with swarms of micro virtual machines in a cloud environment. As the worlds data centers grow larger, so does the focus on energy consumption. This thesis presents a prototype of an architecture which allows for virtual serial communication. Through full feature and synthetic testing it was shown how this prototype could potentially be a communication channel for managing swarms of large virtual machines over a virtual serial interface. The experiments were conducted in a real-life production environment, showing stable results when communicating with potentially thousands of virtual machines. Further improvements of the tools has been investigated with the aim to increase the performance even more.

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I would like to thank my supervisor Alfred Bratterud for giving me the opportunity to work on such an exciting topic for my thesis, it has truly been a pleasure. His technical insight, contributions, and vision has been inspiring.

Next I would like to thank Kyrre Begnum for his unlimited dedication, enthusiasm, creativity, and his ability to see solutions where others only see problems.

In addtion I would like express my sincere appreciation to my family and friends who have stood by me through not only this thesis, but the whole masters program. I am forever gratefull for all your love and support. I would especially like to thank some of my closest friends for sticking by my side and backing me up when things were at the most chaotic.

Last but not least I would like to thank Thomas Hage and Ane Oland for housing me for last 2 months of this masters program. It has been a sincere pleasure.

Chapter 1

Introduction

1.1 Problem domain

Cloud computing is increasingly becoming a bigger part of businesses and the private world. A study published in 2013 estimates that the worldwide cloud computing market will have a 36% annual growth rate through 2016 [14]. Amazon offers one of the biggest cloud computing services on the market today. AWS host services for big international companies such as Netflix, Spotify, Reddit, and Adobe amongst many others. Netflix is presumably one of the biggest streaming sites today, and in 2010 Netflix moved its streaming service into the AWS cloud.

The first six months of 2013, the average enterprise budget for cloud computing was 8,234,438$. This means that the average enterprise would have to increase its budget with 151.54% within the end of 2016. Even though this would profit the companies on some level, it is still a severe increase over a short period as 3 years[14].

A traditional data center is often referred to as housing computer systems and related equipment in a facility. This facility will hold physical machines along with communication devices, storage, powersupplies etc. A cloud on the other hand is a more abstract concept. A cloud consists of many physical machines connected in order to share resources in real-time. The physical machines will act as a bottom layer where a cloud technology running on top will connect the machines into a cloud. Normally the cloud will have a virtualization technology such as KVM or XEN.

The power usage of the biggest data centers around the world is often not disclosed. The reason for this may be not to share information concerning energy usage with competitors, that the results are not what they were estimated, or perhaps that the energy used comes from less green environments. However, in 2011, Google openly discussed the power usage of their data centers around the world, and revealed that they continuously draw 260 million watts. This is about the same amount as a quarter of the output of a nuclear power plant. Every time a search is done or a Youtube video is being watched, these data centers use electricity.[18]

On a global basis, the estimated electricity demand has essentially remained unchanged the past three years. Despite this, data center electricity demands globally had an increase of 19% in 2012 alone. This means a 19% increase of the estimated 31GW demand [13]. In other words, the energy consumption is growing along with the global expansion of cloud services.

When talking about the costs and energy usage, this applies to the data centers which run the IAAS, being one of the main categories of cloud computing. IAAS is an abbreviation for Infrastructure as a service and is the denomination of virtual machines, storage and networks. IThese are the services that are sold to clients, normally paid on a "per-use" or hourly basis. The clients have no responsibility of maintaining or running the equipment, this is all taken care of by the organization which offers the IAAS. Virtual machines are normally priced by virtual hardware defined such as memory, storage, and cpu.

Virtual machines(vm) are software emulations of hardware. A virtual machine has the same functionality and possibilities as a physical computer. VMs can run either directly on top of a hypervisor, or on top of a host operating system which again runs the hypervisor. A hypervisor is software that runs one or more virtual machines. The hypervisor will make sure that the virtual machines get their own environments that is isolated from the host operating system as well as other virtual machines. The use of virtual machines allows servers to run multiple operating systems, which again allows for applications to run in parallel.

A virtual machine without an operating system is just a mapping from software to hardware. The virtual machine is not consuming energy before an operating system is present. In most cases a virtual machine needs an operating system to run a service or a process. Even when the operating system is idle and not performing any tasks, the internal clock is still running and the operating system is consuming energy. In other words, it is the operating system itself which is resource consuming, not the virtual machine.

Todays virtual machines in cloud environments are performing numerous services around the world. Some of these services are running non-stop, and some are only performing services a short period of their uptime. Even though the machines are not performing services or computations, the operating systems idle state is still consuming resources. The machines processors still wake up multiple times per minute to check if there exist any tasks to be performed. This leads to unecessary power usage. Unecessary power usage again leads to more expences and costs.

For example, the smallest VM option available at Amazon EC2 is an Ubuntu instance with 615 MB memory and a 7.9GB disk image[15]. What if a service running on this instance does not require this much resources? What happens to the resources that are not in use? They are being wasted. If a virtual machines purpose is to provide one single service to its users, then the machine‘s resources should be focused on that service.

A lot of the focus towards operating systems has been on making them smaller, more resource effiecient or lightweight. Even though this is a ongoing trend, it is all based on traditional operating systems, such as Linux or Solaris. This is a step in the right direction, but perhaps not the right solution. By using micro virtual machines, which are using the bare minimum of resources, the idea is to make resources focus on the application or service. There is no need for the virtual machine to waste resources on the operating system. The virtual machine itself will practically be just a service.

A challenge for system administrators is that by the use of extremely small operating systems, traditional system administration tools will no longer apply. Tools such as monitoring software or configuration management systems will have to change the way they interact with the virtual machines. Services like remote access, ssh etc will probably not be avalaible since these tools will expect a full size operating system.

Communication is not management itself, but prerequisite. In order to have proper management, a good standard on how the communication process is carried out is crucial. Even though management is the focus of this paper, the necessity of a solid communication layer is essential.

As a result of this, the achievement of really resource efficent machines will cause a tremendous shift in the field of system administrations way of thinking. Furthermore, there exists no design today for how such solutions can be implemented to current cloud solutions. The path is unclear as of how to move to modern cloud infrastructures, towards the capabilty of hosting and managing large deployments of microscopic virtual machines. More research is needed on how a swarm-like group of microscopic virtual machines, can be attempted to be managed in order to truly reduce the energy footprint of the cloud services.

1.1.1 Problem statement

Problem statement: How can we manage the communication with swarms of micro virtual machines in a cloud environment?

"*When I give a minister an order, I leave it to him to find the means to carry it out*." -Napoleon Bonaparte

In the same way customers approach cloud companies and describes what they want, how it is implemented does not concern the client. Different clients will describe different scenarios and problems, but the main principle is the same. Describing a general management interface protocol would profit both hypervisors and virtual machines. It is a two way relationship. Normal, steady and robust channels for management communication do not apply when groups of micro virtual machines grows to 20000, 50000 or even a 100000 machines. By applying a new protocol for this type of communication it will result in many benefits concerning remote management.

An interesting perspective to this is what information from the virtual machines would be interesting to the hypervisor and vice versa. The usage of important information could help optimize the environment the virtual machines are for both themselves and the hypervisor. A hypervisor has no way of knowing how well a service is performing on a virtual machine. What if the virtual machine itself could tell the hypervisor or broadcast to the rest of the virtual machines of its current performance and, by doing this, be placed on the server which will optimize its own and others services? Conversely, the same principle could be applied for virtual machines not having any traffic or very little. This sort of information could be useful on many levels.

By reducing the clock rate or completely turning it off, it is possible to reduce the power usage of a virtual machine to a bare minimum. The micro vritual machine[15] has its clock completely turned of and the clock will not cycle unless the virtual machine is woken up. For example, when attending a conference and staying at a hotel, you don't want to be late for early morning appointments. So you call the reception and order a wake up call. The same principle applies to any operating system and the hypervisor. A virtual machine could call the hypervisor(reception) and order a wake up call. The virtual machine would then be woken by the hypervisor and offer its service.

The hypervisor could ask for a status report on how the virtual machine is performing. Based on statistics on Network, IO, CPU, and RAM usage, the virtual machine can evaluate its own performance so the hypervisor can evaluate whether the virtual machine should be able to migrate or not. This could possibly also be a decision made by the virtual machines themselves by comparing their own performance to other virtual machines.

As Openstack offers a cloud service it is also responsible for some infrastructure services for its clients. By infrastructure services, this projects defines these as SSH and VNC. These are services the cloud should, in some way, want to guarantee for in order for clients to be able to maintain and manage their virtual machines. To protect privacy, an SLA could exist in detail of what services the hypervisor is allowed to inquire about.

Chapter 2

Background

This paper focuses on exploring the possibilites in managing the communication of virtual machines over virtual serial interfaces in a cloud environment. Therefore it is important to cover the aspects and technologies that play a part in the making of the prototype. This is what this chapter intends to do. By covering these topics, this will hopefully give the reader a better understanding of what challenges that lies to ground of this project.

2.1 Cloud computing

The serial interface is a communication channel that is not as much applied today as it was in earlier years of computer- science. As virtual machines appeared, this was one of the components which was inherited and became part of a virtual machine instance. Even so, there has not been many new ways on incorparating the virtual serial interface as relevant factor in virtual machine management. Neil Armstrong once said that "*research is creating new knowledge*". The purpose of this chapter is to look into what foundation this thesis is built on and how it will be integrated in the experiments in order to reach a possible solution for this project.

The concept behind this thesis could have been explored through simulation or a physical server running standard virtual machines. Instead, this thesis seeks to explore how virtual serial communication would apply in a cloud environment. This section will cover virtualization as a concept as well as applied virtualization services. It will also give a brief introduction to vendor specific technologies and the technoligies used for this project such as KVM, Qemu, and libvirt. The section will end with covering Openstack as a service and explain the architecture behind this cloud technology.

2.1.1 Virtualization

In 1967, the first hypervisor was created by IBM. The year after they developed the CP-67 which introduced a new way of sharing resources between virtual machines, more specific RAM. The CP-67 enabled memory sharing which allowed virtual machines to allocate their own memory space. It has been over 40 years

since the virtualization technology was first devoloped and in the recent years, more and more infrastructure is based on virtualization.[19]

Virtualization is a term which can be applied to *hardware*, *storage*, *networking*, *operating systems*, and other commonly used technologies such as emulating a phone for development and testing. Virtualization relies on what is called a *hypervisor*. A hypervisor allows mulitple operating systems to run and share resources on a single host. The hypervisor makes sure that the resources allocated for the different virtual machines are isolated and does not conflict with each others resource space. Hypervisors is often differentiated between *type-1* and *type-2*.[19]

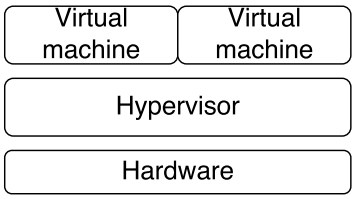

Figure 2.1: Hypervisor type-1.

Type-1 hypervisors can also be referred to as bare metal hypervisors. The hypervisor can directly control the hardware in order to manage the guest operating systems runnning on the virtual machines.

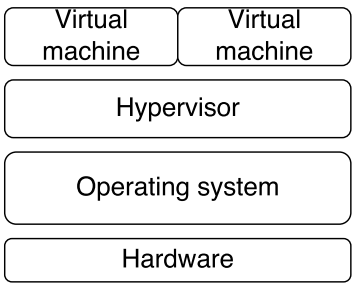

Figure 2.2: Hypervisor type-2.

Type-2 hypervisors runs as a second level software on top of the host operating system. The hypervisor will communicate with the host operating system which again controls the hardware.

Different vendor technologies are often classified as either *type-1* or *type-2* hypervisors. Examples of *type-1* vendors are XenServer, VMware ESX/ESXi, Microsoft Hyper-V, and Oracle VM Server for x86. *Type-2* vendors that may be mentioned are VirtualBox, VMware vSphere, and KVM. At the same time, it is not always easy to differentiate these hypervisor implementations. KVM can be

described as a *type-1* hypervisor since it is applied as a kernal module for Linux and allows its host operating systems to act and use resources as it was running on bare metal. At the same time, different Linux distributions are fully standalone operating systems which then would categorize KVM as a *type-2* hypervisor.

2.1.2 Benefits of type-1 and type-2

There is a distinct difference between *type-1* and *type-2* hypervisors. They both have pros and cons. Since *type-1* is a bare metal hypervisors it will communicate directly with the hardware and therefore reduce overhead. Since *type-1* runs directly on the hardware it also supports hardware virtualization apposed to *type-2* which performs software virtualization. A *type-2* hypervisors normally runs on top of a operating system which has other tasks running alongside the hypervisor. The operating system therefore has distributet its resources, and cannot fully dedicate its resources to virtual machines running in the hypervisors. Because of this, *type-1* hypervisors are most often run in production environments instead of *type-2* hypervisors.

2.1.3 KVM

KVM is an abbrevation for Kernel-based Virtual machine, and is a kernel module which converts the Linux kernel into a bare metal hypervisor. As of the Linux kernel 2.6.20 KVM was accepted into the kernel and therefore benefits from the Linux kernel features. A virtual machine running in the KVM architecture, is running as a normal Linux process. This goes for all the virtual CPU's, and these procesess are scheduled by the Linux Scheduler. [12]. By the use of KVM, virtual machines are allowed access to hardware features of the processor. The virtual machines running as ordinary Linux procesess are running in user-space.

When the KVM kernel module is implemented in the Linux Kernel, all core functionality of the Linux Kernel that applies to memory, process handling, scheduling, etc will be applied to the virtual machines run by KVM. As mentioned earlier, KVM can be viewed as both a *type-1* and *type-2* hypervisor, this all depends on the eye that sees. This debate has arisen since KVM became a part of Linux and many discussions concludes in KVM not falling under either of the classifications.[3]

2.1.4 Qemu

Qemu is an open source machine emulator. Qemu translates the code which runs on the guest os and executes it on the host os. Since it can run unmodified operating systems it can be viewed as a virtual machine monitor or hypervisor. Qemu can emulate several different CPUs(x86, PowerPC, ARM and Sparc). By emulating a complete and unmodified operating systems in a virtual machine where a Linux process is compiled for one CPU, it can be executed on a different CPU.

2.1.5 libvirt(virsh)

Libvirt is an open source API and management tool for virtualization technologies. Libvirt functions as a layer on top the hypervisor and manages the hypervisor through interfaces like virsh. Virsh is the command line interface for libvirt. Libvirt supports several different hypervisors, so the user only has to use libvirt and not be concerned about what hypervisor is running on the lower level. [4]

Virsh itself can eiter be executed directly using virsh –some_command, or by just typing virsh, the prompt will return in a virsh environment where commands can be run.

Example commands virsh

```
#virsh create test_machine.xml

#virsh list

Id Name            State
----------------------------------
1 test_machine     running

#virsh destroy test_machine

### Will delete the guest named test_machine
```

Below is an example output from a virtual machine created using libvirt. The XML describes the devices that should be included on the virtual machine upon creation. This data will exist in a configuration file which is passed as a parameter when running virsh. After a virtual machine has been created, the XML of the virtual machine can be extracted by using **virsh dumpxml**.

Example XML notation used by libvirt

```
<devices>
  <emulator>/usr/bin/kvm</emulator>
  <video>
    <model type='cirrus' vram='9216' heads='1'/>
  </video>
  <disk type='file' device='disk'>
    <driver name='qemu' type='raw'/>
    <source file='/home/alfred/MicroMachines/microMachine.hda'/>
    <target dev='hda' bus='ide'/>
    <address type='drive' controller='0' bus='0' unit='0'/>
  </disk>
  <controller type='ide' index='0'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
  </controller>
  <memballoon model='virtio'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
  </memballoon>
  <interface type='network'>
        <source network='default'/>
  </interface>
</devices>
```

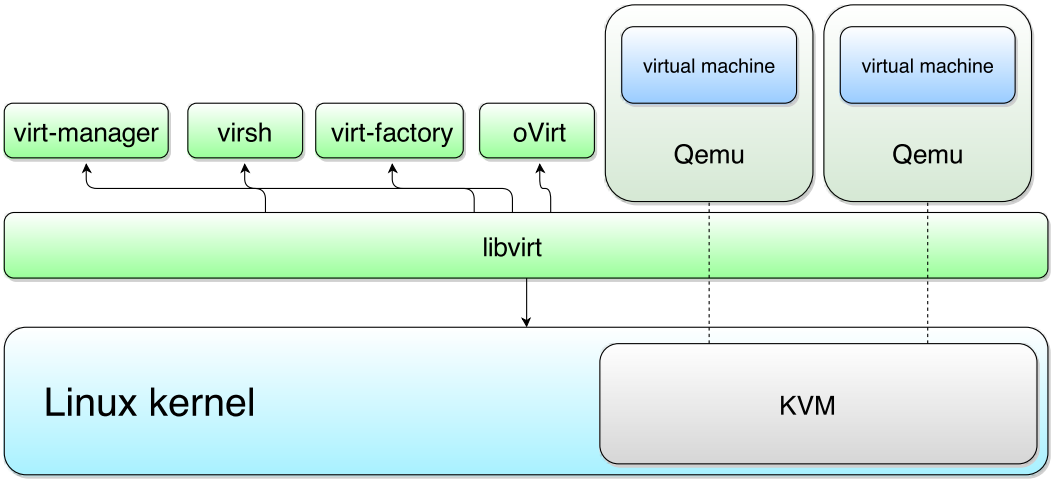

Figure 2.3: Architecture of Qemu, Kvm, and libvirt.

2.1.6 Openstack

Openstack is a cloud operating system which is used as a platform in both private and public clouds. Openstack consists of networking-, compute-, and storage nodes that are managed through a web interface. Openstack supports and uses libvirt as a layer above the hypervisor. The hypervisor used in this project is as mentioned earlier KVM. Openstack consists of several different services, examples mentioned below:

- **Nova compute:** Managing and controlling virtual machines in the Openstack cloud.
- **Neutron:** Handles Networks and IP addressing.
- **Cinder:** Block level storage.
- **Swift:** Object storage.
- **Horizon:** Dashboard web interface for managing Openstack.

Nova compute is the fabric controller of Openstack. Nova is written in Python and its purpose is to manage and control large pools of virtual machines and their needed resources. In other words it manages and provisions networks of virtual machines. The compute arcitecture is designed scale horisontally which means to add more nodes to an application or system. Vertical scaling means adding more resources to an already existing single node in a system. An example where Nova compute comes into play is when a virtual machine is either started or stopped through the Horizon web interface.[7][6]

For network management, Openstack has a system called Neutron. Neutron manages all networks and IP addresses in Openstack and makes sure that there exists a scalable approach to manage all networks. Openstack support different networking models as well as standard separation of traffic through VLANs. Through Neutron, own networks can be created and determine traffic control to and from one or more networks. Also supported is software-defined networking as well as framework extensions for additional network services.[7][6]

Openstack operates with web interface for administrating the cloud. Horizon is a graphical dashboard which interacts with Openstacks resources and can be intergrated with third party applications.

Openstack provides several APIs for development. To make use of these APIs, there has to exist endpoints. The endpoints must be defined in the configuration. A simple model is described in figur 2.4.

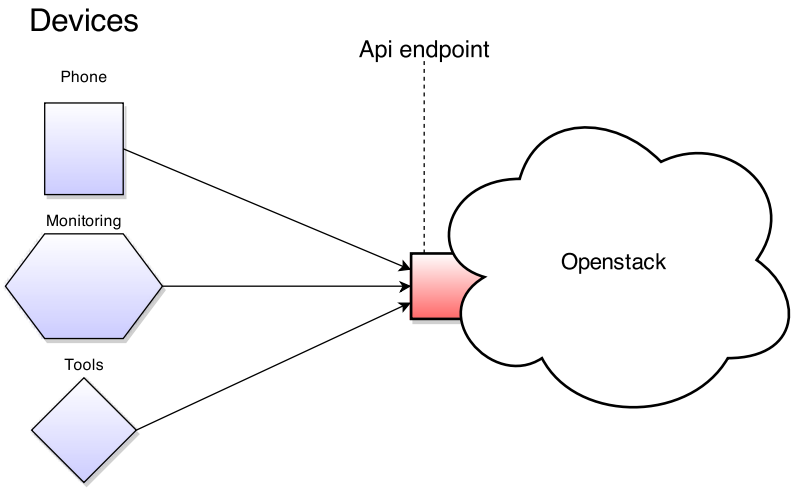

Figure 2.4: Model of communication to Openstack API endpoint.

Endpoint example taken from openstack.com

```
"endpoints":[
    {
        "adminURL":"http://166.78.21.23:8776/v1/604bbe45ac7143a79e14f3158df67091",
        "region":"RegionOne",
        "internalURL":"http://166.78.21.23:8776/v1/604bbe45ac7143a79e14f3158df67091",
        "id":"221a2df63537400e929c0ce7184c5d68",
        "publicURL":"http://166.78.21.23:8776/v1/604bbe45ac7143a79e14f3158df67091"
    }
```

2.2 Message queuing

A message queue provides asynchronous communication between a sender and a receiver. This implies that a sender does not have to interact with a message queue at the same time as the receiver and the other way around. This provides a dynamic channel for communication between a sender and a receiver. A message queue works in the manner of storing a message until it is retrieved. Message queues are applied both in operating systems as well as standalone applications. In both proprietary and open source message queues there are numerous options that may be set in regards to the process of message passing. This could be things such as the durabilty of a message staying in the queue, delivery policies, if messages should be responded with a reciept etc.

2.2.1 RabbitMQ

RabbitMQ is an open source message queue application that uses AMQP(Advanced Message Queing Protocol). This protocol uses the terms *publisher* and *consumer* for what is mentioned earlier as sender and receiver. The publishers creates messages and place them in the message in an *exchange* which routes the messages to the correct queues. The consumers then pick up the messages and process them. To make sure that the message from a publisher reaches the correct consumer, there is need for a message broker such as RabbitMQ. If there is no specific exchange declared for the outgoing message it will be put in the default exchange. [9] [11]

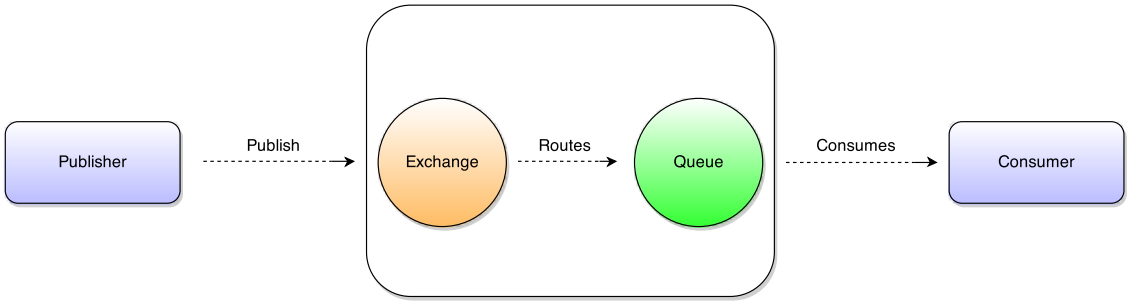

Figure 2.5: Model of communication publisher and consumer in RabbitMQ.

2.3 Communicating with virtual machines

2.3.1 RS-232

RS-232 is a serial interface protocol and describes the standards for serial link communication. The RS-232 can either use synchronous or asynchronous signals when communicating signals. For synchronous communication the sender and receiver shares a clock which dictates the time a bit has been sent. The clock provides information of the next byte being sent over the serial link so both sides are aware of the timing. Asynchronous signaling delimits start and stop of any

byte of data by voltage changes. Along with framing and parity bits, the receiver will sample the voltage level of the byte the number of discrete bits it expects the byte to have. The reciever applies a clock to measure the data elements of the transmission[17].

The Universal Asyncronous Reciever-Transmitter or UART is a standalone chip or part of an intergrated circuit which is used serial communication. The UART converts a parallel data stream of bytes into a serial format of single bits. This means when a serial communication line has been established, the bits are being transferred 1 bit at a time. When the data stream reaches the recipient, the bits are converted back into parallel data for the cpu to use. The UART breakes down the byte before transmission and reassembles it on the recipients side. This process is called framing. In this process a start and either 1 or 2 stop bits are added within the frame along with a parity bit.

2.3.2 Bson

Bson stands for Binary Json and is a binary format for transmitting data. Data is stored as key/value pairs in single entities which are called **documents**. Bson is lightweight, easy to traverse, and indexes very fast. Bson is also very quickly convertable to programming language's native formats. Along with these elements, bson also provides extensions to Json.[2]

Example of bson representation.

```
1 {"hello": "world"} => "\x16\x00\x00\x00\x02hello\x00
2                        \x06\x00\x00\x00world\x00\x00"
```

2.3.3 Virtual serial interfaces

Virtual machines emulates physical machines with all hardware, including serial ports, also known as COM ports. In Unix and Linux operating systems these are mapped to /dev/tty. TTY stands for *teletype* which originated from electro-mechanical machines which where connected through a network over the world in order to transfer commercial telegrams. A tty in itself is more or less an interface that reads and writes text, in other words a native terminal device. Pty stands for *pseudo terminal device* and is emulated by another program for instance xterm. These interfaces work in pairs where one is a slave and another is the master. The master is a file descriptor used by a process which is connected to the slave part of the pty which is what is called pts.[10]

2.4 Related work

This thesis explores the possibility of combining different aspects of computer science into a prototype which will prove a concept. Many of these aspects has

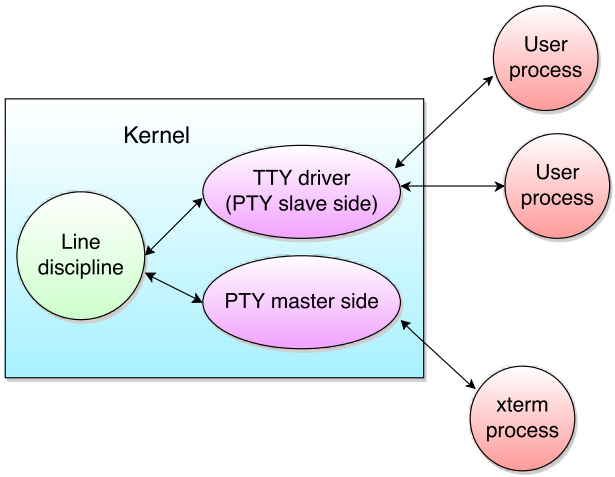

Figure 2.6: TTY interaction with Linux kernel.

been covered well in different papers, but there is a lack of papers which is directly linked to this exploritory thesis. Even so, the related work selected for this project presents and discusses many relevant topics which can be used for the development of a prototype.

2.4.1 Maximizing hypervisor scalability using minimal virtual machines

The work of this project builds on certain aspects of the findings of Alfred Bratterud and HårekHaugerud. They have explored the possibilities of maximizing the scalability of hypervisors by using minimal virtual machines.[15] This project builds on extremely small instances of virtual machines that only consist of 512 bytes. By stripping and reducing the size of virtual machines this project was able to run over 110000 instances on a single hypervisor. The internal clock of the micro virtual machines was completely turned off in order to make the CPU totally idle. The minimization of virtual machines drastically reduces the energy usage and therefore gains potential resources. The reduction in virtual machine size will also able more instances on a physical machine. The project concludes that the reduction of resource usage in operating systems has great potential for reducing RAM and CPU for cloud hypervisors. The work of this thesis builds on certain aspects of this project.

2.4.2 Self Adaptive Particle Swarm Optimization for Efficient Virtual Machine Provisioning in Cloud

This project discusses the possibilities of implementing an effecient virtual machine provisioner which satisfies customer needs and at the same time minimizes power consumption of servers. Often found in cloud computing is the on-demand access to resources. The datacenters running clouds are therefore often over provisioned in order to handle unexpected workloads. The problem in this scenario is to be able to handle virtual machine requests and at the same time reduce power consumption of extremely large and dynamic datacenters. To find a

optimal permanent solution would take a very long time. In the cloud however, it is sufficient to find a **near** optimal solution, preferebly in a short period of time. This project suggests meta heuristic methods such as *ant colony optimization*[1] and *particle swarm optimization*[8] to solve this challenge. This idea is related to this thesis in the context of optimizing power consumption and better the quality of virtual machine services through reporting on the quality they are served by the physical servers.[20]

2.4.3 Virtual Machine Packing Algorithms for Lower Power Consumption

VM-based flexible capacity management is an approach to pack virtual machine on physical machines(PM) in order to reduce power consumption in data centers. This project deals with the problem of data centers not utilizing their power consumption. This problem is addressed by introducing two different VM packing algorithms, respectively macthing-based(MBA) and greedy-heuristic type(GREEDY). The paper discussed problems related to these algorithms, mainly the trade-off between power-saving and user experience, decision on VM packing plans within a feasible calculation time, and collision avoidance for multiple VM live migration processes. In the experiments conducted, the results shows a reduction in power usage between 18% and 50% in total. These results can be related to the thesis by applying the idea of the virtual machines letting the hypervisor know how well it is performing at a certain time compared to previous time. The hypervisor may then take it into consideration when deciding which VMs that should be live migrated. In addition one might consider the idea of the virtual machines giving themselves low priority on a task that is being performed such as backup. The hypervisor could then reevaluate the virtual machines location based on the priority level of the task being perfomed. [21]

2.4.4 A Case for Fully Decentralized Dynamic VM Consolidation in Clouds

In this project, it is proposed a fully decentralized dynamic VM consilidation(VMC) schema based on an unstructured peer-to-peer (P2P) network of PMs. This is done through validating the schema using three well known VMC algorithms. These are respectively First-Fit Decreasing (FFD), Sercon, V-MAN, and a novel migration-cost aware ACO-based algorithm. The project proposes these algorithms to pack virtual machines periodically in idle times to the least number of PM's. The main idea is that this will increase the efficiency in energy usage in data centers. [16]

2.4.5 Self-Adaptive Management of The Sleep Depths of Idle Nodes in Large Scale Systems to Balance Between Energy Consumption and Response Times

This project proposes a sleep state management model to improve energy usage as well as response time of awakening virtual machines. The idea is to group virtual machines based on a sleep level. The sleep depth of the virtual machines classifies

them into reserve pools with different readiness levels. The pool with the highest readiness level will preferebly be chosen towards an application. This approach showed that the power consumption of idle nodes can be reduces by as much as 84.12% with slowdown level of only 8.85%.[22]

Chapter 3

Approach

The approach chapter aims to provide the reader with a detailed plan of how design, implementation, and experiments are performed. This chapter will contain sections with specific details on how the prototype will be developed. The results presented in the next chapter will be based on the plan designed and presented here.

3.1 Approach overview

Since this thesis is an exploritory thesis, there is need for an approach to create a functional prototype. There are many ways to come up with a plan, but in order to find the one which will guide the thesis closest to the problem statement, more than one should be evaluated. It is important to stay focused on the goal at hand and build the the plan around the problem statement. The approach chapter will describe a specific plan to help reach a solution to this project's problem statement: *How can we manage and communicate with swarms of micro virtual machines in a cloud environment?*.

This chapter will state the design, requirements, and process of this project. It will go into details about choices of design, the procedure on how the project will be performed in order to reach the intended problem statement, and levels of architecture protocol handling. The project will be divided into three different phases denoted below which also will be discussed in further detail later on:

- **Phase 1:** Exploring possibilities for two-way serial communication.
- **Phase 2:** Describe architecture, robustness, and flexibility.
- **Phase 3:** Determine and describe functionalities of the protocol itself.
- **Phase 4:** Development of prototype and testing

3.2 Exploration phase

In phase 1 the idea is to map the potential technical opportunities and possibilities within Openstack. This will more or less be a pilot study to determine what possibillites lies within Openstack. By exploring the possibilites of two-way serial communication between a hypervisor and a virtual machine the idea is to choose the most applicable one for this project. Important aspects that will be uncovered in this phase are:

- How Openstacks hypervisors handles virtual serial port mapping to virtual machines through its tenants.
- How virtual machines running in Openstack can read and write to a serial interface.
- Create a working environment where a demonstration can be placed in order to state functional serial communication between Openstack and a virtual machine.

To build a software that can handle two-way serial communication between a hypervisor and virtual machine it is crucial to explore how Qemu itself and the operating system for this project handles virtual serial interfaces. Even though this project builds on the research of micro virtual machines[15], this project will make a proof of concept statement which possibly later on can be implemented in other projects. Therefore the experiments that will be conducted and discussed in this theses will mainly be based on Linux or more specificly Ubuntu 12.04.

Based on the fact that the experiments will be conducted on Linux operating systems it is decisive that there will be a thorough exploration on how Linux handles ttys, also known as virtual serial interfaces. As mentioned in the previous paragraph this also needs to be done with Qemu, since in fact it is Qemu that is running the virtual machines. When these aspects have been properly explored separately it is important to make them function together.

A set of tests will be set up with different scenarios, in order to identify what measures need to be taken for data to be written and read over a virtual serial interface between a hypervisor and a virtual machine. By doing these exploratory tests, it will help discovering the best way to implement a possible solution towards the problem statement.

The exploration phase will end when a conseptual prove has been made and it is possible to demonstrate serial communication between a hypervisor and a virtual machine.

3.3 Design phase

The architecture should have certain properties that fulfills certain demands. This will **not** concern the contents of the messages passed to and from the virtual

machines, only the structure of the passing of information. There are a set of important properties that should be discussed in regards to the architecture:

- Scalability.
- Possibilities for intercommunication between virtual machines.
- Reliable transmission.

After phase 1 and successfully demonstrating serial communication between a hypervisor and a virtual machine, phase 2 will begin. Phase 2 which will be the design phase, will create an architecture of communication in a cloud insfrastructure. This architecture will describe the logistics on how the communication will take place. As phase 1 explores the possibilities for serial communication between a hypervisor and virtual machine, phase 2 will build on these findings and model an architecture sustainable for a swarm of virtual machines. If phase 1 describes the principle, phase 2 will describe the "postal office".

3.3.1 Parallelizing and load balancing

By using the analogy "postal office", the idea is to build the architecture around a sentralized queing system. All traffic passing through the system will go through a "postal office", which will keep track of all messages going to and from the virtual machine. The use of a queing system will be very benifitial for a system that needs to manage traffic to and from a swarm of virtual machines because this will enable asynchronous communication.

In order for this protoype to support a certain level of parallelization, a queing system will be implemented. The queing system will be running on a virtual machine and handle queue all traffic going to and from the compute nodes. By having a sentralized queue, multiple units can produce data to the queue simultainously. As mentioned this will allow for asynchronous communication. In other words this implies that messages can be produced to the queues without any device at the same time constantly consuming from the queue. The message that are to be consumed from the queue will remain there until they are collected. This allows messages to be written to the queue at any time it suits the sender. This will create the possibility for a large scale of virtual machines constantly writing to queues at any given time, opose to a service permanantly has to listen for incoming traffic.

By the use of a sentralized queue, this will create an aspect of loadbalancing. All the messages will be stored in a queue until somebody is ready to collect it.

In addition to the ability to send messages to virtual machines from a client node, the virtual machines themselves should be able to establish communication and send messages over the tty's to and from each other. This applies for both virtual machines running on the same compute node as well as virtual machines running

on different compute nodes. A potential design will be created for how this will work, but not be implemented in the testing of the protoype. The prototype will test communication from a seperate machine running a script which will send messages to the queueing server. This machine could simulate any machine or actor which has an intention of communicating with virtual machines based on this projects architecture.

3.4 Implementation phase

This section will describe how the design will be implemented in a given environment. It will include software, scripts, datastructure, and how these elements will be combined in order to later perform tests. A prospect will be presented on the total architecture of the design.

3.4.1 Implementation environment

As decleared in the problem statement, the attempt to manage and communicate with a swarm of virtual machines, is to be done in a cloud environment. The current environment this experiment is going to be preformed in, is a production environment used at Oslo and Akershus University College of Applied Sciences. This is a fairly new data center currently holding 11 compute nodes and runs Openstack as a cloud service. The reason for conducting experiments and building the implementation in this environment will give this project the opportunity to achieve the best possible results based on the problem statement.

Even though the cloud is currently part of a production environment, it is possible to create isolated environments which will be suitable for this project. The idea is to create a testbed with a sufficient amount of virtual machines to support the problem statement. These machines are to be configured and equipped with the necessary software before booted up. This implies that they are listening and are able to receive traffic on the correct ttys as well as being able to write to them.

3.4.2 Simulation

This project is reaching to implement a prototype of a protocol that may create the possibilty to make use of the ttys between virtual machines and a second part. An alternative to implementing the solution in a cloud environment is to perform simulations of the entire project and collect results respectively. This would give an impression of a completely isolated environment, which potentially could give a false impression of accomplishment. As a real production environment is available, this will give the the most realistic results.

3.4.3 Software

Figure 3.1 is a rough outline of how the final design will be. The controller and compute node will be part of the Openstack cloud. The controller in this case

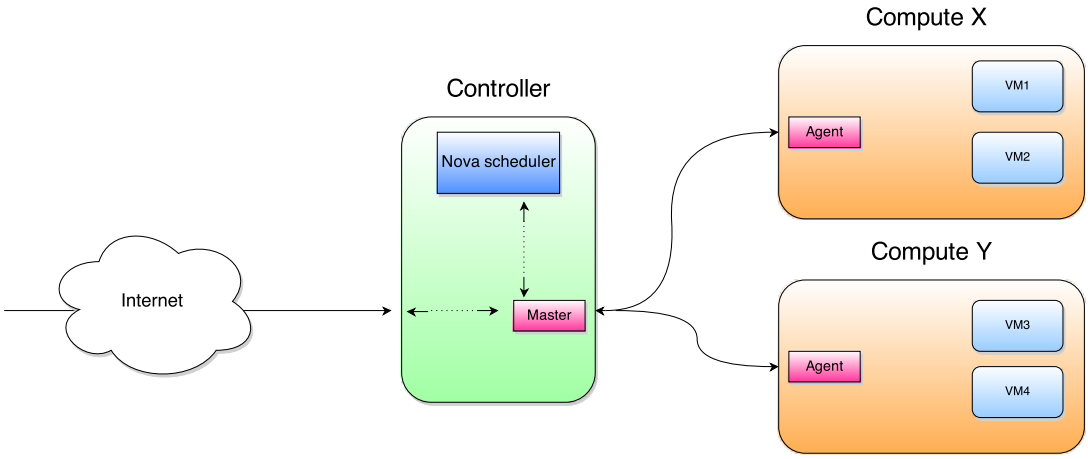

Figure 3.1: Prospect of how protype will be set up.

both represents the RabbitMQ as well as Nova as a interactive part. The reason for this is that Nova as an example could interact with the virtual machines as well as standard client. The cloud image to the left in figure 3.1 represents the potential client(s).

As figure 3.1 has been noted and described in the last paragraph, a more detailed description of the individual elements of the prototype can now be explained. There will be four major elements which influence the how the prototype will function. They are as follows:

- The client tool.
- RabbitMQ
- The compute node service.
- The service running on the vms.

In order to interact with virtual machines and send messages, there is a need for a client process which will act as an interface to the user. This interface should be able to broadcast a specific message or send it to a single virtual machine. To make sure that traffic can be sendt to a specific virtual machine or a number of machines a script needs to collect data about the virtual machines running in Openstack. This data should be collected through the Openstack database. Data concerning the virtual machines in addition to physical should be collected in order to build a datastructure for messaging.

The datastructure will consist of the virtual machine's UUID, compute node which holds the virtual machine, a shipment ID, a return queue, and the message itself. The shipment ID will make sure that the client service is able to recognize

the message for the current session running. The return queue is a part of the data structure in order for the compute node to be aware of where the answer from the virtual machine should be published. And finally, the message itself. This datastructure should be constructed for all messages going out from the client side.

After the datastructure is created, the client script should connect to the RabbitMQ server and created outgoing queues as well as a return queue. These outgoing queues should be created based on the compute nodes that holds the respective virtual machines that are to receive a message. In addition to being able to publish messages to outgoing queues, the script has to consume responses from the return queue and return them.

The process running on the compute nodes which contains the virtual machines should build on the findings of the exploration phase. The process should be able to handle traffic to and from multiple virtual machines simultainiously. This could possibly be done either by forking a process into multiple processes, or having a process create threads to handle every virtual machine. Since every virtual machine has their own tty connected to the hypervisor, there may have to be a thread listening on both the hypervisor and the virtual machine. This is an important aspect concerning scalability. Both parts should be able to recieve traffic at the same time as they might be initiating the traffic. As well as handling traffic to and from the virtual machines, the script running on the compute nodes has to connect to the queueing server in order to consume and publish messages.

Finally, a last script is needed on the virtual machines themselves. The intention of this is to read and write data to and from the virtual interface or tty. All outgoing and incoming messages to a virtual machine will come from the tty. After reading a message from the tty the virtual machine will process it, possibly execute it, and create a response if necesarry. The response will then be written to the serial interface and further handled by the compute node. The case will be the same if it is the virtual machine itself which initates traffic.

3.5 Testing

When a conceptual prove has been made, of how two-way communication over a tty works, a test environment will be set up. This environment will consist of virtual machines preconfigured and implemented with the necessary software to perform the experiments. The virtual machines will be spread over a set of compute nodes. There will be performed both real tests as well as synthetic. Since the cloud used for this project will not support running swarms of virtual machines, some of the tests will be as mentioned synthetic and will only simulate certain scenarios which will be used for collecting data as reference points.

A synthetic test will be performed in order to test what effect the queueing system will have on the architecture. Messages will be created and placed in the correct queues as normal, collected and read, but forwarding to the serial interface will be simulated. The reason for this is as mentioned that the cloud will not

support as many virtual machines as needed to become a swarm. This type of synthetic test will only apply when the amount of messages is increased to imitate traffic to a swarm of virtual machines. Prior to this test, a measurement will be made to estimate the time a compute uses to write a message to the tty and read the response. A compute node will, in this scenario, read the messages from a queue, sleep the amount of time it uses to write to and read from the tty, and write a simulated response back into the queue. In this case, it will be possible to see how the implementation of RabbitMQ affects the architecture when thousands of messages needs to be handled.

Finally, a test will be performed in order to show how sustainable the architecture will be when multiple messages will be sent to the virtual machines running in the test environment. Pending the achievement of being able to handle a stable two way communication to a different tty's, this test will attempt to prove the idea of communicating with multiple virtual machines over the virtual serial interface or tty.

3.6 Alternative approaches

To prove the concept that there exists a way of communicating and managing virtual machines through virtual serial interfaces there are alternative approaches. As mentioned earlier, simulation was a possibility, but the selected approach was to implement the solution in a cloud environment existing at Oslo and Akershus Univeristy College. Another implemtation environment could be a public cloud such as Amazons EC2. Amazon EC2 is one of the biggest cloud services on the marked today.

Since this project focuses on a proof of concept, another possible approach could be building the exact same implementation, but leaving out the ttys. Instead of traffic passing over the ttys this could be implemented with TCP or UDP. Even though the whole project is based on the usage of virtual serial interfaces, it is the concept which is important. This implies that the same concept could be proven with the use of normal networking traffic such as TCP or UDP.

3.7 Expected results

The project is divided into 4 different phases that together complete a protoype of alternative communication protocol. In order to say that a possible solution to the problem statement is reached depends on the prototype. The exploration phase is expected to give answers to how a communication channel works over tty's in a virtual environment. The answers given here will be critical on how the design will turn out in the end. If the design and implementation succeeds in the preferred way, the final results will be evalutated in the perspective of how well this is suited for managing virtual machines. It is expected that the final implementation will give the possibilty for both real and synthetic testing.

3.7.1 Approach summary

This chapter has proposed the reader with an approach to how the prototype will be developed. The approach has been divided into four different phases where three of them describes the different build aspects of the protoype, and the last and final phase explains how the testing is intended to be performed.

- **Exploration Phase:** Exploring possibilities for two-way serial communication.
  - How Openstacks hypervisors handles virtual serial port mapping to virtual machines through its tenants.
  - How virtual machines running in Openstack can read and write to a serial interface.
  - Create a working environment where a demonstration can be placed in order to state functional serial communication between Openstack and a virtual machine.
- **Design Phase:** Describe architecture, robustness, and flexibility.
  - Design a working blueprint of the architecture.
  - Focus on asynchronuous communication.
  - Focus on scalability.
- **Implementation Phase:** Determine and describe functionalities of the protocol itself.
  - Build working tools and services based on design.
  - Implement the blueprint.
- **Development of prototype and testing:**
  - Design and conduct both real and synthetic testing based on design and proof of concept.

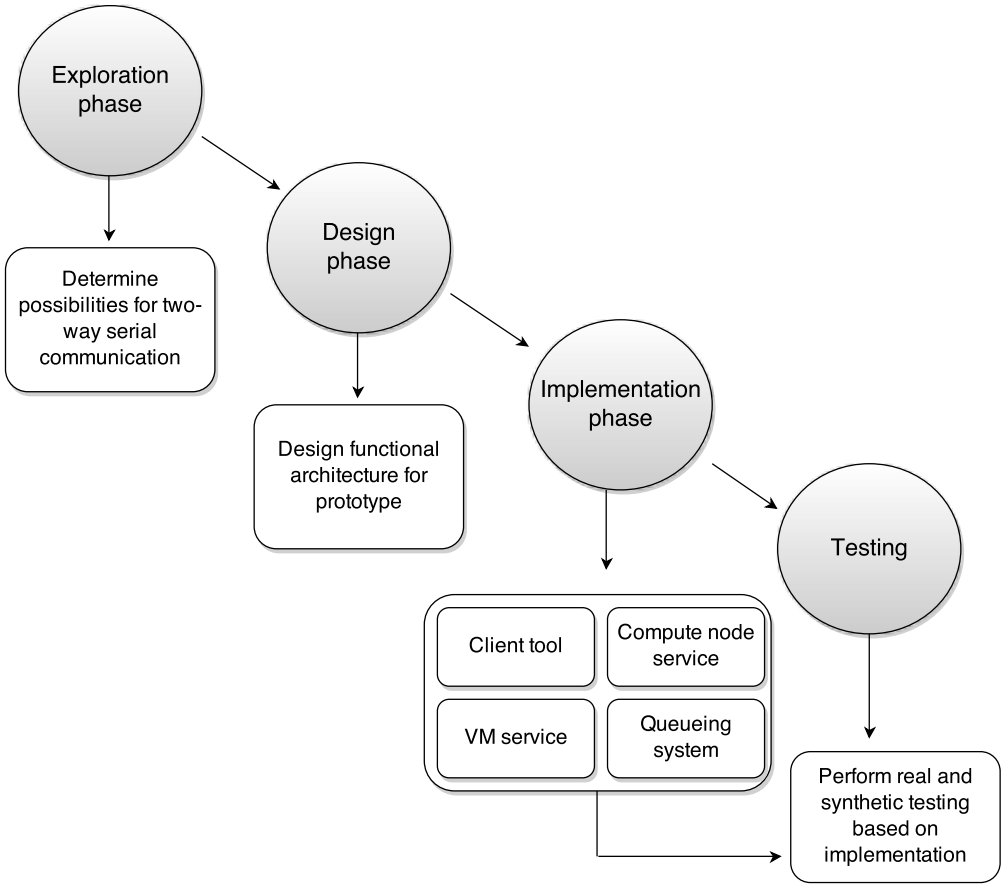

Figure 3.2: Approach overview.

Chapter 4

Results

This chapter will examine and present the results of the findings of the four phases of this project. Each phase will be presented separately in their own subsections.

4.1 Exploration of tty communication

This section will show how it is possible to establish a to-way communication scenario by using the virtual serial interfaces between a hypervisor and virtual machine running in an Openstack cloud. This proof will be used in order to create a functional prototype of the communication protocol. The tests and results presented in this phase are more preliminary prior to the larger experiments.

4.1.1 Introduction to virtual serial communication

Preliminary setup The first experiments were done on a KVM/Qemu hypervisor running a virtual machine with Ubuntu 12.04. This was set up in a cloud environment with a single VM. The problem statement seeks to find out how this can be done with a swarm of virtual machines, but for the experimentation phase a single virtual machine was sufficient. The exploration phase seeks to establish a toway connection over the serial interface between a hypervisor and virtual machine. This setup would create the foundation for later testing and experiments.

Virtual serial interfaces in Linux are denoted as *tty* or *pts*. These are located under /dev/ttySx or /dev/pts/x where x is a number. These interfaces are virtual character devices that provides a bidirectional communication channel. In other words, they are pseudoterminals. The channel is divided into two parts. One side of the communication channel is called a *master* and the other part of the channel is called a *slave*.[5]

The *master* - *slave* relationship between the pty's is shown in figure 4.1. The master side of the character device is the one which initiates or creates the pty. In other words, the process or network device which "owns" the pair. The slave side

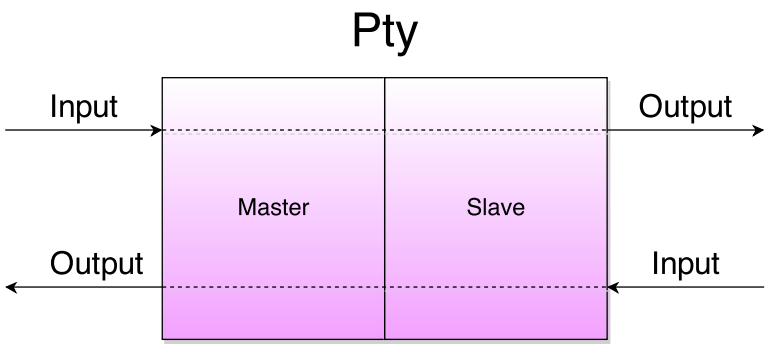

Figure 4.1: Description of pty. Master-slave.

is for other procesess or devices which hooks up to the communication channel. When a pty is opened the kernel creates a file handler for the pty so it can be written to read from as a file. As a single pair running in one kernel, it is a fairly easy task to read from and write to. The complexity level increases when it is being combined with transmission over a virtual serial interface to another master - slave pair running as a virtual machine with its own kernel.

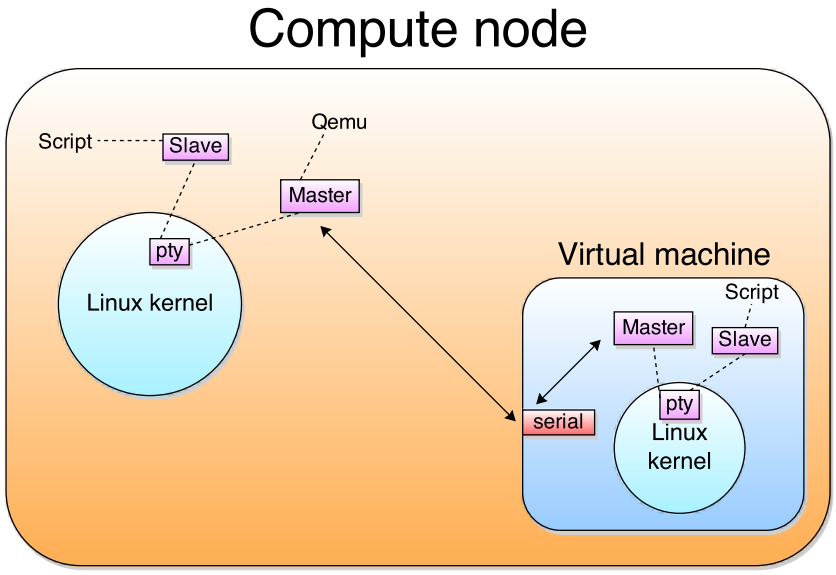

Figure 4.2: Pty devices in virtual machine run in Qemu

Figure 4.2 shows how the mapping of pty/ttys works over a virtual serial interface. As the virtual machine is run inside the hypervisor, there now exist a two master - slave pairs to the same pty. The communication channel now flows through 2 kernels between one pty in each end of the channel. One on the hypervisors side and one on the virtual machine side.

4.1.2 Communication channels

Since this was done in a virtual environment where KVM/QEMU was running the virtual machine, the preliminary experiment had established how data is written to a pts on a hypervisor, passing through KVM/QEMU, and ending up at the ttyS1 on the virtual machine. The virtual machines running on the hypervisor was managed by libvirt and could therefore be interacted with through the command line interface virsh. Virsh has a command that allows to dump an xml description of virtual machines. All virtual hardware details about a virtual machine is stated in the xml. In the xml dump there exists a line which contains information concerning which pts is connected to ttyS1 on the virtual machine. An example of the xml notation is shown i the background chapter under section 2.1.5.

Part of xml dump from libvirt: serial interface

```
1  <serial type='pty'>
2      <source path='/dev/pts/3'/>
3        <target port='1'/>
4        <alias name='serial1'/>
5  </serial>
```

The dump above is collected by performing the command **virsh dumpxml** and supplying either **<instance-name>** or **<UUID>** of the virtual machine as a paramater to dump the information. As seen on line 2 of the dump above the source path of the pty is **/dev/pts/**. This is the serial interface on the hypervisor which is connected to ttyS1 on the virtual machine running on the hypervisor.

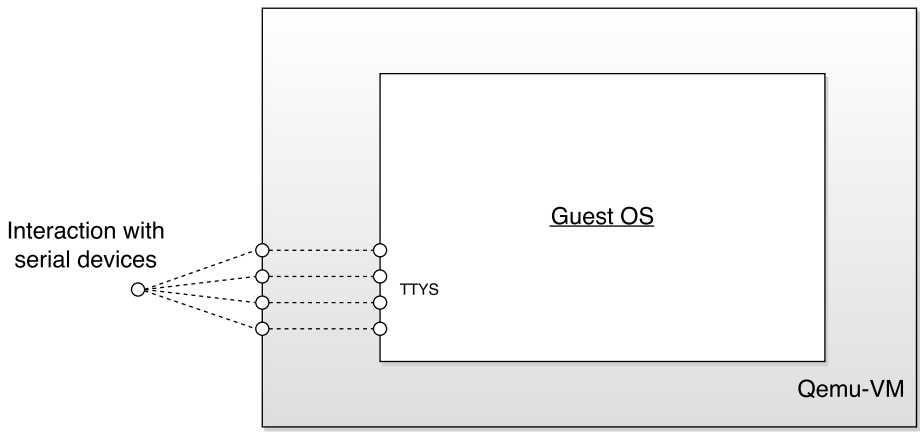

Figure 4.3: Serial interface setup on a Qemu virtual machine.

4.1.3 Preliminary testing of communication over virtual serial interfaces

The section will describe results from the preliminary tests in tables. The tables consist of the order of the communication, which device performing which task, which task is being performed, and whether or not the communication attempt was successful. The respective columns will be filled with either a checkmark or a crossmark. In addition some comments will be made on the individual table. The tests were performed by echoing a string into the virtual serial interface and by listening with the shell command **cat** on the other side of the channel. In order to state that a complete two-way connection has been established, three scenarios has been decleared and needs to be fulfilled:

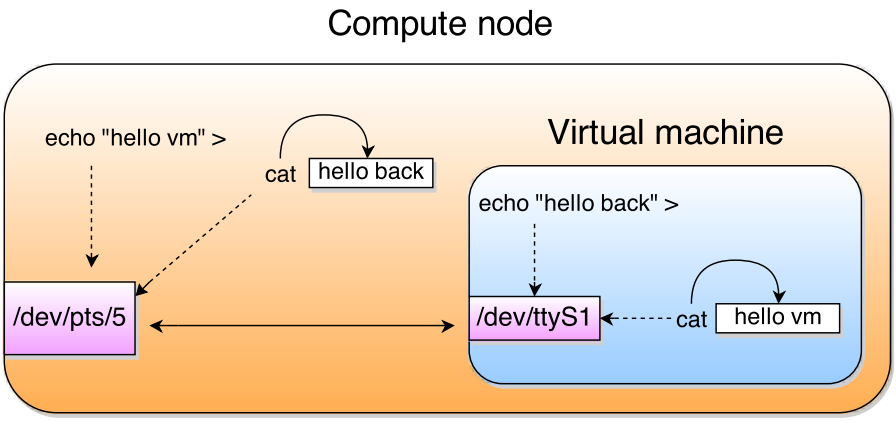

Figure 4.4: Simple description of preliminary tests.

- Sc_1 - VM writes to tty and hypervisor recieves the data.
- Sc_2 - Hypervisor writes data to the pts and VM recieves the data.
- Sc_3 - Two-way communication is established. Data can be sent and a response can be written and received at the sender.

The three first tables will represent scenario Sc_1,

Order	Device	Writing	Listening	Working
1	VM	✓	✖	✖
2	Hypervisor	✖	✖	

Table 4.1: Results from preliminary test 1, Sc_1.

Table 4.1 describes the first attempt of writing to the serial interface from the VM to the hypervisor. The VM in this case will perform the command **echo "something" > /dev/ttyS1**. This test is denoted as test 1 from scenario Sc_1. In this test the virtual machine wrote to ttyS1, without the hypervisor listening.

Order	Device	Writing	Listening	Working
1	VM	✓	✖	✖
2	Hypervisor	✖	✓	

Table 4.2: Results from preliminary test 2, Sc_1.

Table 4.2 shows the results of test 2 from scenario Sc_1. The setup is similar as in test 1, but this time the hypervisor is listening. The listening process starts after the string is written to the tty. The process of the table would then be as follows:

1. The virtual machine performs the command: **echo "something" > /dev/ttyS1**
2. The hypervisor performs the command **cat /dev/pts/3**. Where 3 is the pts connected to the ttyS1 on the VM.

As the listening process starts after the writing, this causes the messages be lost.

Order	Device	Writing	Listening	Working
1	Hypervisor	✖	✓	✓
2	VM	✓	✖	

Table 4.3: Results from preliminary test 3, Sc_1.

Table 4.3 shows the results of test 3 from scenario Sc_1. Notice that the order has changed. Prior to the virtual machine writing, the hypervisor is listening on its tty, in this case it was /dev/pts/3. The data was recieved and collected by using the command **cat /dev/pts/3**. The difference in in the process compared to table 4.2 is:

1. The hypervisor performs the command **cat /dev/pts/3**. Where 3 is the pts connected to the ttyS1 on the VM.
2. The virtual machine performs the command: **echo "something" > /dev/ttyS1**

Since the listening process using **cat** started prior the the writing process, it was now possible to collect the bytes written to ttyS1.

Sc_1 has described how data is written and read going from a VM to a hypervisor. The upcoming test is denoted as test 1 from scenario Sc_2. In this test the hypervisor wrote to /dev/pts/3, without the virtual machine listening.

Order	Device	Writing	Listening	Working
1	Hypervisor	✓	✖	✖
2	VM	✖	✖	

Table 4.4: Results from preliminary test 1, Sc_2.

The process of table 4.4 consists of the hypervisor performing the command **echo "something" > /dev/pts/3** and the virtual machine remaining idle.

Order	Device	Writing	Listening	Working
1	Hypervisor	✓	✖	✖
2	VM	✖	✓	

Table 4.5: Results from preliminary test 2, Sc_2.

Table 4.5 shows the results of test 2 from scenario Sc_2. The setup is similar as in test 1, but this time the virtual machine is listening. The listening process starts after the string is written to the tty /dev/pts/3 on the hypervisor. The process is then as follows:

1. The hypervisor performs the command: **echo "something" > /dev/ttyS1**
2. The VM then performs the command **cat /dev/ttyS1**. Where ttyS1 is connected to the /dev/pts/3 on the hypervisor.

Order	Device	Writing	Listening	Working
1	VM	✖	✓	✖
2	Hypervisor	✓	✖	

Table 4.6: Results from preliminary test 3, Sc_2.

The results of Sc_1 and Sc_2 has so far been similiar, except for test 3. In this case, even if the VM is listening on ttyS1 prior to the writing to /dev/pts/3 of the hypervisor, the string does not reach the virtual machine.

Order	Device	Writing	Listening	Working
1	Hypervisor	✓	✓	✓
2	VM	✖	✓	

Table 4.7: Results from preliminary test 4, Sc_2.

After test 3, Sc_2 was performed it was learned that in order for data to be written from the hypervisor to the virtual machine, the hypervisor has to listen simultaneously. If the hypervisor is not listening at the same time, the virtual machine will be blocking when trying to read its ttyS1. The virtual machine will be blocking until the hypervisor listens to the connected pts.

1. The hypervisor performs the command **cat /dev/pts/3** in order to listen prisor to the writing of the message. This is because the VM will be blocking when trying to read ttyS1, if the hypervisor not simultaneously reads /dev/pts/3.
2. The hypervisor performs the command: **echo "something" > /dev/ttyS1**
3. The VM then performs the command **cat /dev/ttyS1**. Where ttyS1 is connected to the /dev/pts/3 on the hypervisor.

Order	Device	Writing	Listening	Working
1	Hypervisor	✓	✓	✓
2	VM	✓	✓	

Table 4.8: Results from preliminary test 1, Sc_3.

Order	Device	Writing	Listening	Working
1	VM	✓	✓	✓
2	Hypervisor	✓	✓	

Table 4.9: Results from preliminary test 1, Sc_3.

The third and last scenario in the exploration phase looked into how data could be written and responded to through the serial interfaces. Based on these tests, a virtual machine can write to its tty and if the hypervisor is listening, the data can be read. This is oposed to the hypervisor which needs to listen when wrtiting in order for the vm to be able to read the data. So if both sides of the communication channel are listening, they both are able to write and read from the serial interface.

4.1.4 Summary of preliminary tests

The preliminary tests has shown how data can traverse both ways over a virtual serial interface. The results have discovered that there exist a prerequisite when it comes to writing to and from a hypervisor to a virtual machine. In order for the virtual machine to be able to read the data written to the tty, the hypervisor has to read/listen to the connected /dev/pts. If the hypervisor fails to do this, the virtual machine will end up blocking until the read is executed by the hypervisor on the correct /dev/pts. In addition there are elements which controls who or what writes to a tty at a specific time. This could potentially cause corrupted messages since bytes written could be mixed.

4.2 Design

The design and modelling aspect of this protocol will be an important one in order to ensure a certain level of quality. This section will present the result concerning architecture choices, modeling, the general build and design of the implementation.

4.2.1 Asynchronous communication

Blocking is the concept of when a process has to wait for a resource, an event or I/O in order to proceed with its execution. If some event is blocking it will stay there until a certain resource has been freed or some event occurs etc. By seeing the blocking scenario in coherence with transferring messages to a swarm of virtual machines this could potentially be a major problem. Blocking should be avoided in order to create an asynchronious communication channel. If a swarm has to wait for 1 single machine because of blocking, the whole protocol will fail. The reason for the blocking may be unknown, the machine may not be powered on, the process handling the messages has crashed etc. There is a definite need for handling this problem within the protocol to make sure each transaction is independent from the others.

4.2.2 Design apllications

The prototype has four major applications concerning virtual machine communication. These four ways of communicating differs in who the initiator is.

- C_1 - Communication initated by **send_msg tool**.
- C_2 - Owner(hypervisor) initiates contact with VM(s).
- C_3 - VM initiates contact with owner(hypervisor).
- C_4 - VM initiates contact with another VM.

C_1 is denoted as a scenario where the initiator is someone executing the script from the client side such as the example in figure 4.9. This could also be a situation where a service within Openstack such as Nova wants to use the communication channel for scheduling virtual machines.

C_1 makes use of the **send_msg** tool which gives the implementation certain properties. The **send_msg** tool will supply the architecture with a set of properties which will allow for possibilites such as asynchronous message checking. **send_msg** will allow the user to produce and publish messages to queues based on which virtual machines that it wants to initate contact with. The tool itself will construct a datastructure based on the VMs the messages are going to. The string sent to a virtual machine will be based on this datastructure and contain elements such as shipment id and response queue id.

With the use of a shipment id and a response queue id, the user can execute the script, send messages, and collect the response(s) at a later point in time. This will allow for asynchronous message checking. The response queue id created for the specific shipment will be returned to the user along with the shipment id. The return queue id can be used to check wheter or not any virtual machines have responded to the messages later on. The reason for returning the shipment id as well, is to make sure the user collects the message(s) belonging to that specific shipment.

The tool will also allow for instant checking of responses. This is done by executing the tool and waiting for a response. The waiting time is set before executing and will denote the maximum time to wait for a response. If the response is received prior to the wating time set, the tool will stop the execution and print the response.

C_2 describes the communication design of compute node initiating contant with a running virtual machine.

C_3 describes the communication design of a virtual machine initiating contact with another virtual machine running on a different compute node.

C_4 describes the communication design of a virtual machine initating contact with another virtual machine running on the same compute node.

4.2.3 C_1

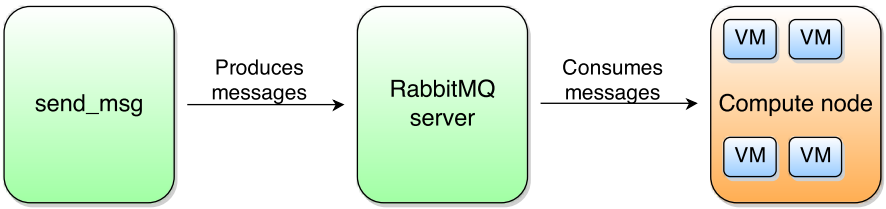

Figure 4.5: Simple overview of send_msg.

Figure 4.5 displays a very simple overview of how **send_msg** interacts with RabbitMQ. More details will be explained in the implementation/prototype section.

4.2.4 C_2

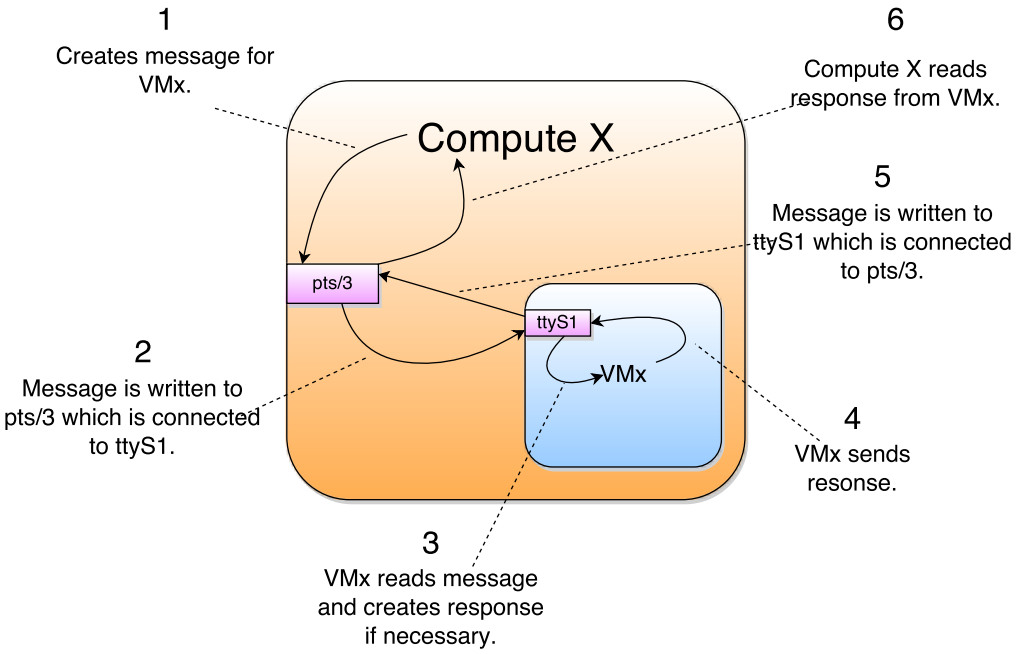

Figure 4.6: Compute node initiating traffic to a VM.

Figure 4.6 describes communication between compute node and a virtual machine running. Even though the figure graphically describes the flow of traffic from a compute node to a virtual machine, the concept would be the other way around. The figure shows 6 steps in order to complete a transaction from a compute node to a running virtual machine within that specific environment. Details of these steps are described below, and the reader is reminded that the transaction process could be turned around.

1. First the compute node creates a message which is intended for a virtual machine running locally on that specific compute node. An example of such a message could be a wake up call or for instance asking if a specific process is running.
2. The message is written to the pty connected to that specific virtual machine which in this case is denoted as pts/3.
3. At the same time as the message is wrtten to pts/3, the virtual machine is listening on ttyS1 which is the virtual serial interface connected to pts/3. When the message comes in, the virtual machine reads and processes the message and creates a response if necessary.
4. If necessary, the virtual machine creates a response to send back to the compute node.
5. If a response has been created, the virtual machine writes this back in to the virtual serial interface ttyS1 for the compute node to receive.
6. At the same time, the compute node listens for incoming traffic on pts/3. If traffic is coming in on the compute node will read and process the response.

4.2.5 C_3

Figure 4.7 describes communication between virtual machines on different compute nodes through virtual serial interface. As seen in figure 4.2, a virtual machine interacts with another virtual machine located on a different physical server through RabbitMQ. All traffic coming and going to and from a physical server is handled by a process running on that specific compute node. A detailed description of the different steps described in figure 4.2 are described below.

1. Virtual machine VMx located on compute node X wants to interact with virtual machine VMy running on compute node Y. So the first step is to create a message it wants VMy to recieve.
2. The message is written ttyS1 which is connected to virtual serial interface on compute node X which is here denoted by pts/3.
3. Compute node X then reads the message, processes it and ships it into the correct queue in order for it to reach its destination at VMy.
4. Compute node Y consumes from the incoming queue specified for itself, processes it, and writes it to the correct pts connected to virtual machine VMy. In this case the pts is denoted as pts/4.
5. The message is written to pts/4 which is connected to ttyS1 on VMy.
6. VMy is listening for incoming traffic on ttyS1 and reads the incoming message. After processing the message it creates a response if necessary.

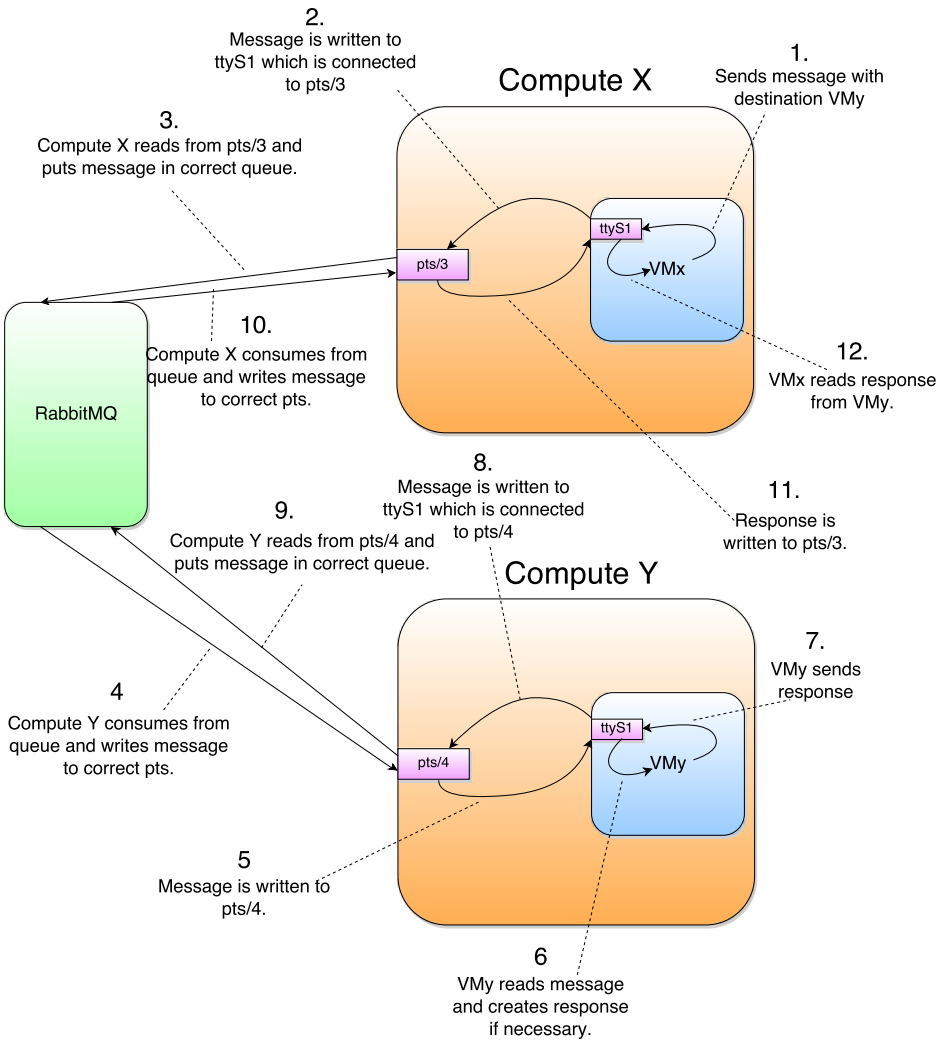

Figure 4.7: VM initiating traffic to VM located on a different physical server.

7. If a resonse has been created, VMy sends the response.

8. The response is then written back into ttyS1 which is connected to pts/4 on compute node Y.

9. Compute node Y processes the message and ships it to the correct queue for compute node X to consume from.

10. Compute node X consumes from the correct queue and locates the correct pts connected to VMx.

11. The response from VMy is written to pts/4 which is connected to ttyS1 on virtual machine VMx.

12. Virtual machine VMx listens on ttyS1 and reads the incoming response and processes it.

4.2.6 C_4

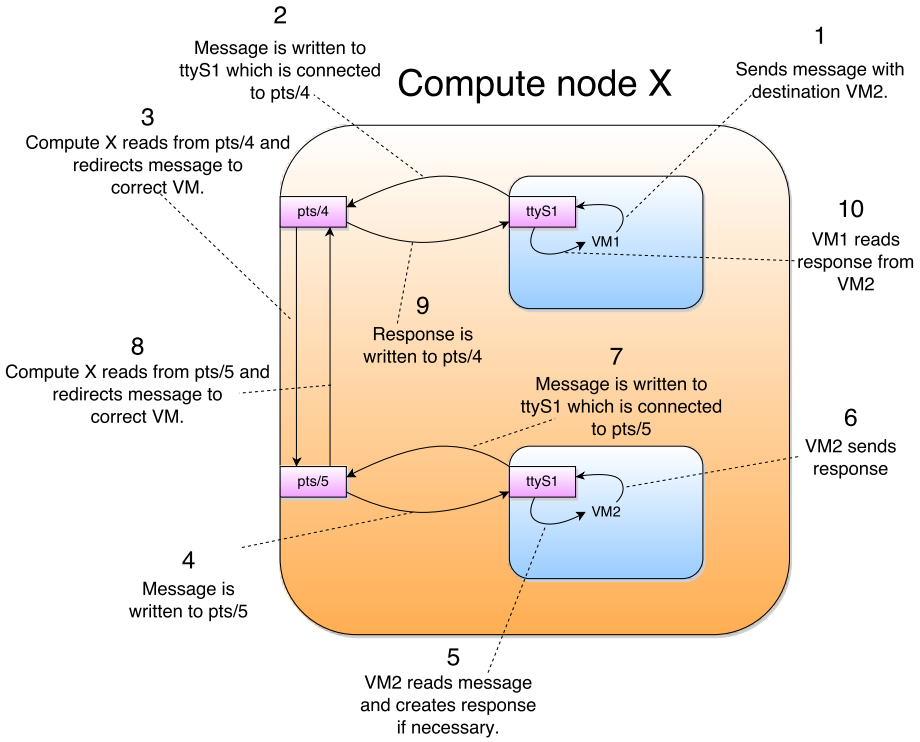

Figure 4.8: VM initiating traffic to VM located on the same physical server.

Figure 4.8 describes traffic between two virtual machines running on the same physical compute node X. As opposed to the traffic going between two virtual machines located on two different pysical compute nodes, the messages sent between them does not need to pass through a queue. A detailed description of the steps are described below.

1. VM1 located on comoute node X creates a message with the destination of VM2 located on the same compute node X.
2. The message is written to ttyS1 which is connected to the virtual serial interface on compute node X which in this case is denoted by pts/4.
3. Compute node X reads and processes the message from VM1 and redirects it to VM2. This is done by simply writing the message to the correct pts connected to VM2. In this case the correct pts is denoted by pts/5.
4. The message from VM1 is written to pts/5 which is connected to ttyS1 on VM2.
5. VM2 listens on ttyS1, reads and processes the message. If necessary, VM2 will create a response.

6. If a response has been created VM2 sends the response back to compute node X.

7. The response is now written to ttyS1 which is connected to pts/5 on compute node X.

8. Compute node X reads and processes the message and redirects it to VM1. In the same way as in step 2 this simply means writing the message to the pts connected to VM1. Which in this case its pts/4.

9. Response is now written to pts/4 which is connected to ttyS1 on VM1.

10. VM1 reads and processes the response from Vm2.

4.2.7 Design summary

This section has described the four different types of communication scenarios. The different scenarios have been described seperately with complete details of the communication processes. Even though the design phase includes four different types of communication over a virtual serial interface, the prototype at hand will only focus on implementing C_1. The tool described in C_1, namely **send_msg** will covered thoroughly in the upcoming implementation/prototype section.

4.3 Implementation/protoype

This section will implement a protoype based on the results from the design phase. The purpose of this section is to thoroughly describe the working process of the protoype, how it is built, and implemented. The implementation and the prototype will be based on communication type 1 denoted as C_1 in the design section of the results.

4.3.1 Serial port locking

To make sure that data written from a hypervisor to a virtual machine, through the virtual serial interface stays uncorrupted, a lock has to be set on the tty. The same principle is also valid the other way around. A mutex lock will make sure that only one of the two parties may write to the tty at the time. The reason for having a mutex is because at any given time the virtual machine may send data to the hypervisor or the other way around. Should there be a conflict and both parts are writing the tty at the same time, the two messages would be written the tty and possibly mixed. So when both parts are attempting to read from the file, neither part will get a correct message, rather just a combination of bytes from the two messages. By implementing a mutex this will make sure there can be only one part writing to a tty at the time, even though this will put the other part in blocking mode until the tty is released. This is of course an important aspect, when a part is finished writing to the tty, it is crucial that it releases the lock.

4.3.2 Message format

The message format is a string which consists of the virtual machine UUID, the response queue, the shipment id, and the message itself. The different elements will be contained in the same comma seperated string. The UUID will be used by the script running on a compute node in order to find the correct tty for forwarding the message to a specific virtual machine. The response queue is used for placing any responses to a message. The shipment can be used as reference to make sure that messages read in the current session actually belongs there. Since both the shipment id and the queue id are generated randomly there is a minimal chance of this actually happening. Finally the message itself, is a normal string.

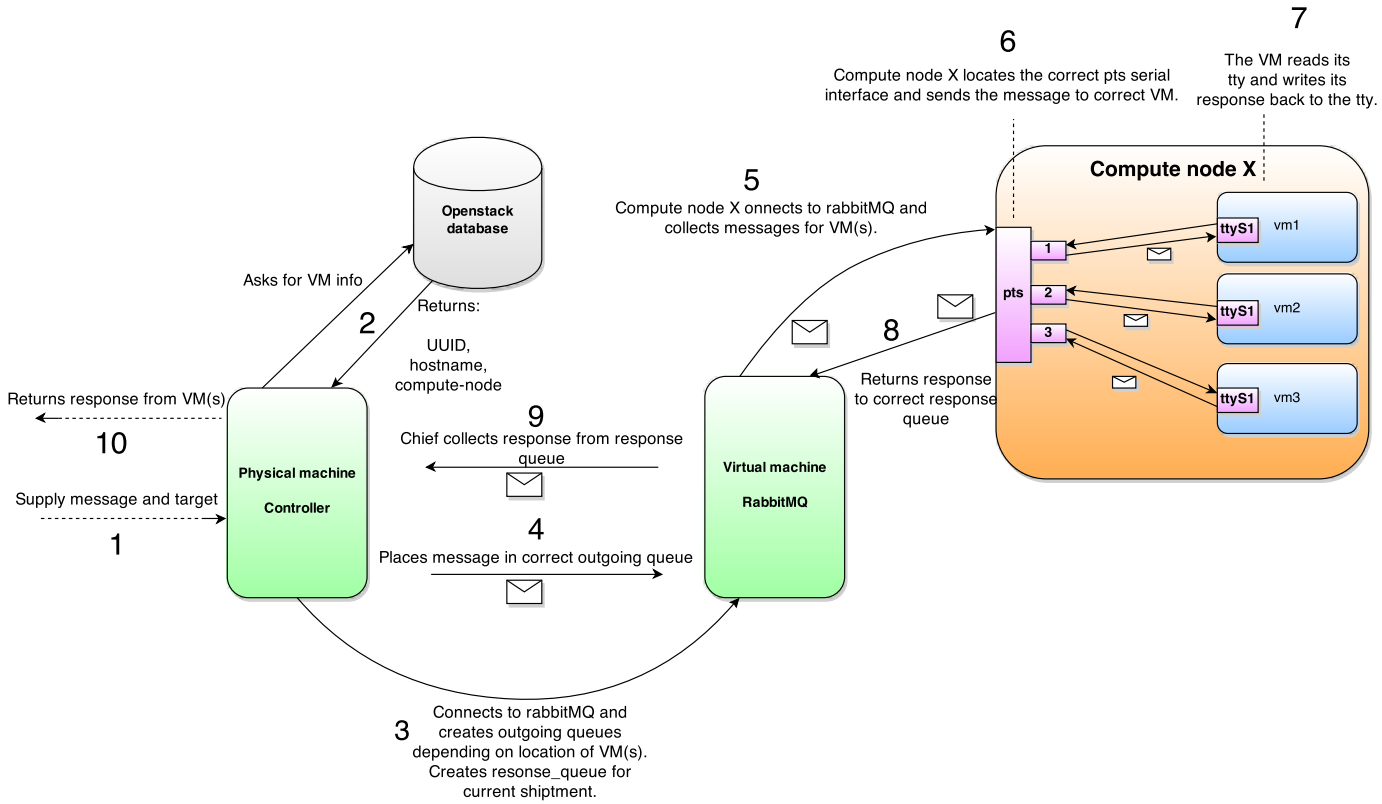

Figure 4.9: Detailed model of implementation. Traffic initiated from a client perspective

Figure 4.9 describes the communication process from a client perspective in order to deliver messages to VMs. The figure describes a step by step process on how the traffic is flowing and what processes that are in action in order to send a message and recieve a response. Below the steps are explained in detail:

1. The first step is determening what message is to be sent and which VMs that should receive the message. Both of these parameters can be set when executing the initiation script.
2. When the script starts it will contact the Openstack database and collects information on the VMs that are receiving the message. The information collected are UUID, hostname and the name of the physical machine the virtual machine is located on.
3. Before starting the actual transmission of the message(s), the script will contact the RabbitMQ server and create outgoing queues for all the physical machines that are to receive messages. The queues are created depending on which VMs that are going to receive messages. In addition, a response queue is created in order to collect responses from the VMs.
4. After the queues have been created the script will build a datastructure for the messages and put them in the correct queues.
5. The compute nodes will now start consuming messages from correct queues and create a number of threads depending on how many VMs are going to receive messages.
6. After a message has been consumed from the queue, the compute node will open the package and check the UUID. This is the identification number used by Openstack for a virtual machine. The compute node will then dump xml data based on that UUID in order to find the correct pts to forward the message to. After collecting the correct pty, the message is written to it.
7. The virtual machine is now listening on ttyS1 and will be able to receive the message. After the message has been processed, if necessary, the VM will create a response and write it back into the ttyS1.
8. Like the VMs the compute node will also listen on the pty to be able to collect resonses coming from virtual machines. When a response is read from the pty, it will be placed back in the response queue in RabbitMQ.
9. The client machine which initated the communication will be consuming from the response queue within RabbitMQ as the responses are coming in.
10. Returns the response(s).

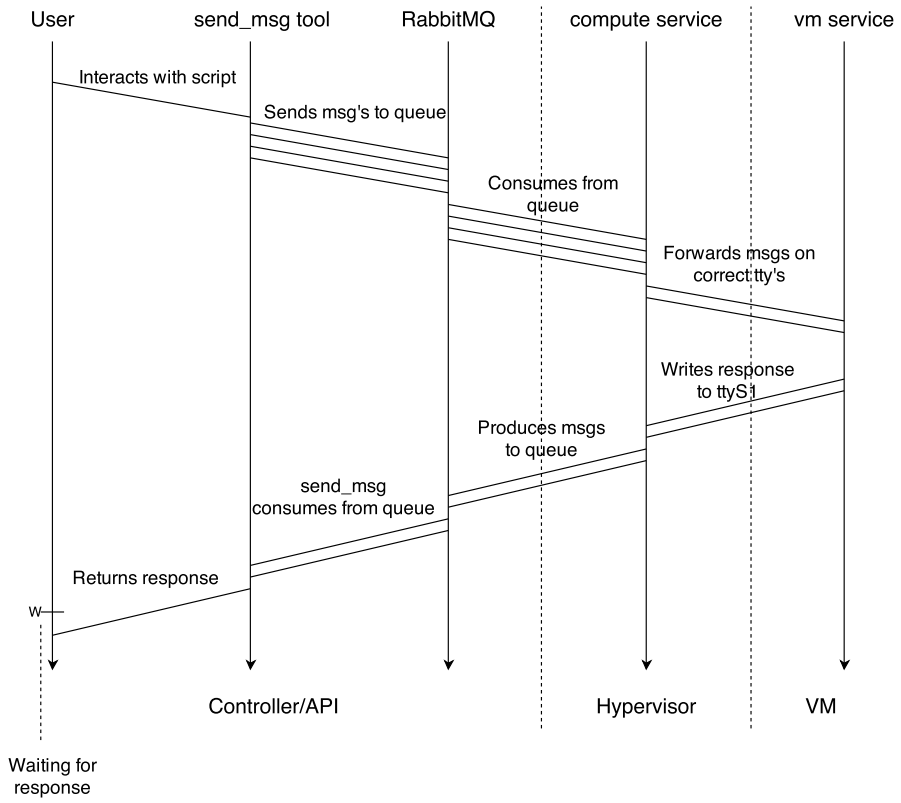

Figure 4.10: Process flow of prototype implementation.

Figure 4.10 describes the process flow of the protoype. The figure incorporates all instances of the flow in order to give some perspective to the process. This is a description of a full round trip of the prototype, where a message is initated by a user, delivered to a vm, and a response is written back and collected.

4.3.3 send_msg tool

Figure 4.9 describes a complete implementation of the protype and its traffic cycle. Step one states that a message along with a target are supplied to a machine running the *send_msg.pl* script. The target is denoted as the virtual machine or virtual machines which are intended to receive the message. This tool works as a gateway for all traffic, and was developed in order to interact and test the protocol from a client perspective. This script will only be running at one machine throughout the testing. The main function of this script is to be able to communicate with the virtual machines without regards of how the serial communication is handled. By producing messages for virtual machines to a queue, they are later collected by a script running on a compute node which forwards the message over a serial interface to the virtual machine.

The *send_msg.pl script* has several functionalities depending on the usage. The different functionality the script holds can be evoked by passing arguments and

switches to the script. Some of these are mutually exclusive which in other words mean that if a specific parameter is passed to the script others may not be applied in the same session. The current available functionality in the script are as follows:

- **<t>** - Argument which denotes the target(s) that will recieve the messages.
- **<m>** - Argument which contains the message to be sent.
- **<w>** - Argument which denotes the blocking wait time. When sending message(s) this is the total time the script will wait for a response.
- **<i>** - Switch which returns shipment ID and queue ID for later use.
- **<c>** - Argument which checks the queue applies for messages.

Example run of send_msg.pl

```
1 ./send_msg.pl -m "command20" -t /serial1/ -w 30
```

Above is an example of a standard run of the *send_msg.pl* script. The script will now pass the message "command20" to a any virtual machine named "serial1", and wait 30 seconds for a response. The script itself will terminate and exit if the response has arrived before the timer has expired.

When *send_msg.pl* starts, the first that happens is that the script will establish a connection to the RabbitMQ server. When the connection is established it will start building a datastructure based on the virtual machine(s) that are to recieve a message. The datastructure will consist of hash which stores information the information listed below:

- UUID of the virtual machine.
- A shipment ID of this current session.
- A queue ID for the current shipment.
- The message itself.

Example of datastructure

```
1 auto75.serialtest
2     UUID: fc1b99c0-6d4f-4a7e-8bca-9104b1dc508d
3     Located at PM: compute08
4     Shipment_ID: 145811
5     Content: fc1b99c0-6d4f-4a7e-8bca-9104b1dc508d:8825147:145811:command20
```

The datastructure is as mentioned in the paragraph above based on the virtual machines which is going to acquire messages. This information is collected from the Openstack database. The script will query the instance table which holds all necessary information about the running virtual machines. The target which is passed as an argument to the script through the switch **-t** <target> can either be a specific virtual machine, or be a mysql regular expression. Based on what is passed as an argument, a specific query will created and executed.

After the Openstack database has been queried, a random shipment id and queue id will be created for the current session. If the random shipment id or queue id contains one or more 0's, these will be exchanged with 1's.

Once the random shipment ID and queue ID have been created, the script will continue and create a response queue based on the random queue ID and outgoing queues based on the location of the virtual machines. The outgoing queues will be named after the hostname of the compute nodes accordingly. Below there is a description of how the sub routine *create_outgoing_queues()* creates outgoing queues on the RabbitMQ server based on the data collected and stored in the hash **%OUTGOING**. This is also described graphically in figure 4.11.

Creating outgoing queues from send_msg.pl

```
1  sub create_outgoing_queues{
2      $CHAN = $CONN->open_channel();
3      my @cnodes;
4
5      foreach my $vm (sort(keys %OUTGOING)) {
6          foreach my $uuid (keys %{$OUTGOING{$vm}}){
7              foreach my $compute_node (sort keys %{$OUTGOING{$vm}{$uuid}}){
8                  push (@cnodes, $compute_node);
9              }
10         }
11     }
12
13     @cnodes = uniq @cnodes;
14
15     for(my $i=0; $i<=$#cnodes; $i++){
16         $CHAN->declare_queue(
17             queue => $cnodes[$i],
18             durable => 1,
19         );
20     }
21 }
```

Subsequently of the creation of the datastructure and the needed queues for the current shipment the script will start producing messages to the respective queues. The script will now call the sub routine *send_msg()* as shown below. This function will iterate through the hash and produce the messages to the correct queues. In addition it will keep track of the number of messages produced.

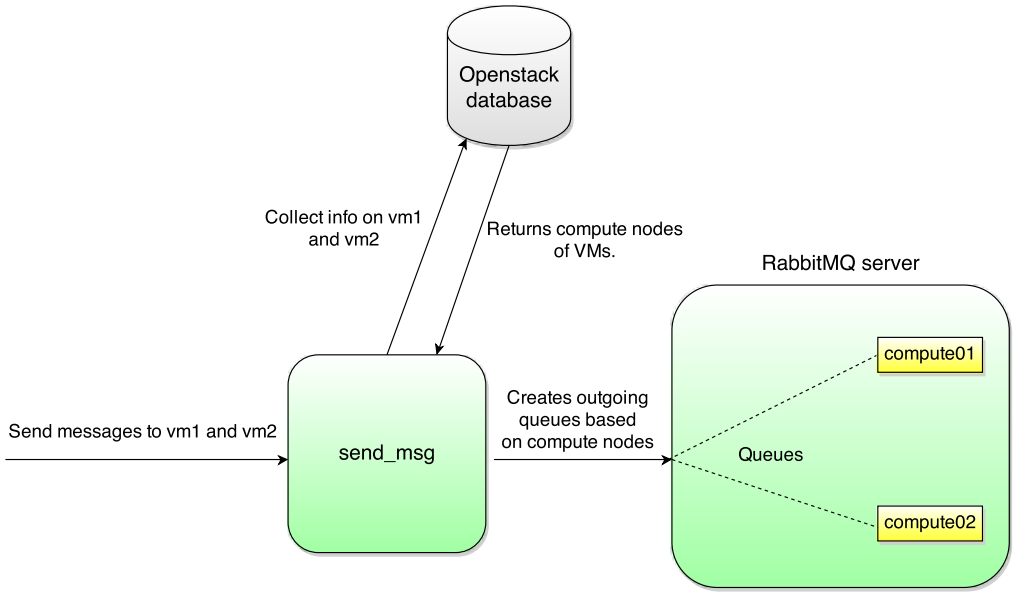

Figure 4.11: Process of queue creation by send_msg.

Send message function from send_msg.pl

```
1  sub send_msg{
2
3  foreach my $vm (sort(keys %OUTGOING)) {
4        foreach my $uuid (keys %{$OUTGOING{$vm}}){
5              foreach my $compute_node (keys %{$OUTGOING{$vm}{$uuid}}){
6                    foreach my $ship_msg_id (keys %{$OUTGOING{$vm}{$uuid}{$compute_node}}){
7                          {$compute_node}{$ship_msg_id}\n\n";
8                          my $string = $OUTGOING{$vm}{$uuid}{$compute_node}{$ship_msg_id};
9                          my $send_msg="$uuid:$READ_QUEUE:$SHIP_MSG_ID:$string";
10                         $CHAN->publish(
11                               exchange => '',
12                               routing_key => $compute_node,
13                        body => $send_msg,
14                         );
15                         $MSG_COUNT++;
16                   }
17             }
18       }
19 }
20 }
```

Depending on the parameters passed to the script, it will act accordingly. Making use of the example stated earlier, it were to wait a total time of 30 seconds for response(s). After the message(s) have been passed to the queue(s) on the RabbitMQ server, the *send_msg.pl* script will now start consuming from the response queue which previously was created. The the script will call the sub routine *read_queue()* from the script. This function will establish a connection to the RabbitMQ server and wait for any messages located in the response queue. If a message appears in the queue, the sub routine will call a callback function which will handle the message and print it to the terminal. In addition, it will keep track of the number of responses collected in order for the session to terminate when the

number of responses equals the number outgoing messages. It will also set a *true* value in the **$EMPT_RES** variable in order to not listen forever when using the switch **-c**.

Callback function from send_msg.pl

```
sub callback {
      my $var = shift;
         my $body = $var->{body}->{payload};
         print " [x] Received $body\n";

         my @c = $body =~ /\./g;
         sleep(scalar(@c));

         print " [x] Done\n";
         $CHAN->ack();
      $RES_COUNT++;
      $EMPT_RES=1;
}
```

4.3.4 Compute service

The compute service is set up to run on all the compute nodes in the cloud environment. Its purpose is to read from the queues and forward messages to the correct virtual machine and process the response. This section will in detail describe the functionality of the compute service and give insight on how it is integrated in the communication architecture.

The compute nodes which are running the *compute* service is somewhat the most important link in this architecture. In order for the messages to reach the virtual machines over a virtual serial interface is through this particular service. It is the *compute* service's job to read and write to and from the correct serial devices based on the virtual machine that is to receive the message.

The *compute* service will listen for any incoming messages that are to appear in its own designated incoming queue. All the compute nodes have 1 specific queue they are to read from. The *read_queue* function on the *compute* service will read from the queue using a callback function.

Callback function from compute.pl

```
sub callback {
      print "TRACE:$VM_UUID:Start callback, reading from queue:" . Time::HiRes::time . "\n";
      my $var = shift;
         my $body = $var->{body}->{payload};
            print " [x] Received $body\n";

      system("/root/kjetil_master/return_answer.pl '$body' &");

      my @c = $body =~ /\./g;
      sleep(scalar(@c));

      print " [x] Done\n";

```

```
14      print "TRACE:$VM_UUID:Finished reading msg from queue:" . Time::HiRes::time . "\n";
15      $EMPT_RES=1;
16      $CHAN->ack();
17  }
```

The callback function will consume the message from the queue and send the message as a parameter to a script which will handle the message. This was intended to run in parallel using threads, but the threading implementation turned out not performing the way it was expected. Therefor a different solution was implemented by calling a second service to handle the message while the *compute* service listened for incoming messages. This created the functionality of parallelization where the main process where listening for incoming traffic, and the second process handled the messaged and died. The service that handles the message is *return_answer*.

When a message is received it processed and split into variables. All that is passed on to the virtual machine is the shipment id as well as the message itself. This done by calling the *construct()* function. After splitting the message the *construct* function will pass the UUID of the virtual machine to *get_tty* which will find the correct pty connected to the virtual machine.

get_tty function from compute.pl

```
1   sub get_tty{
2       if($SIM){
3           open(VIRSH,"virsh dumpxml e606cfc4-3900-4cd9-b429-d47dae526179 |");
4       }
5       else{
6           open(VIRSH,"virsh dumpxml $VM_UUID |");
7       }
8       while(my $line = <VIRSH>){
9           if($line=~/<serial type=\'pty\'>/){
10              $line=<VIRSH>;
11              if($line=~/<source path='(\/dev\/pts\/\d+)'\/>/){
12                  $TTY_PATH=$1;
13
14                  last;
15              }
16          }
17      }
18      close(VIRSH);
19  }
```

The *get_tty* function makes use of the *virsh* command in order to get the correct serial device to forward the message to. The UUID is used by virsh to dump xml data of the virtual machine, and a regular expression collects the correct serial device which is stored $TTY_PATH.

After the *construct* function has been called, the message written to $TTY_PATH which contain the correct serial device or pts. The writing process is done by calling the *write_tty* function. After the message is written, the service will read the serial device twice. The reason for this is that the compute will read the

string already written, and it is important that this is not considered as the response. Therefore the *read_tty* function is called twice and the correct answer responded by the virtual machine is collected.

When the correct respense has been collected, it is the service's job to publish the response to the correct response queue. The correct response queue is a part of the message when it is first consumed from the queue designated to the compute node.

4.3.5 VM service

This section will describe in detail how the VM service works in this communication architecture. The section will look into how messages are handled when they reach a virtual machine and how responses are written back. This is an important part of the prototype and has been placed on running virtual machines used for testing the implementation

All the virtual machines which was a part of the testing was supplied with a script called *vm.pl*. The main intention of this script was to read incoming messages from the tty connected to the hosting compute node, process the message, and write back a response. The tool is built to handle recognized commands and call a respective sub routine based on the command. If the command is not recognized, the virtual machine will respond with a "command not found" string.

Hash of functions from vm.pl

```
1  my %SUBS = (
2          "command1" => "action1",
3          "command2" => "action2",
4      );
```

The commands are stored in a hash where the command name is the key and the function name is the value. Below is an example of how a sub routine is called from a hash.

Hash of functions from vm.pl

```
1  $SUBS{$function}->().
```

In order to read from the tty on the virtual machine a perl module called use *Term::ReadKey* was used. The *read_tty* function reads the content from tty until a 0 appears. The string passed to a virtual machine through a serial interface(tty) contains the shipment id and the message itself. The function opens the ttyS1, reads the bytes, and splits the string on ":". The shipment id and message are stored in global variables which are utilized later on.

Read tty function from vm.pl

```
sub read_tty {
   open(TTY, "<$TTY_PATH");
   print "Waiting for message: ";
   ReadMode "raw";
   my $string;

   while ( my $key = ReadKey 0, *TTY ){
      $string .= $key;
   }
   ReadMode "normal";
   close(TTY);
      my @content=split(":",$string);       #splits message
      $SHIP_ID="$content[0]";
      $VM_MSG="$content[1]";
   print "Got string: '$string'\n";
   return $string;
}
```

The *listen_tty* function below reads ttyS1 and checks wether or not the message is and existing command on this virtual machine. If it is, it will execute the command through a sub routine and write the return value back to ttyS1.

Listen function from vm.pl

```
sub listen_tty{
      while(1){
            my $msg=read_new_tty();
            chomp($msg);
            foreach my $function (keys %SUBS){
                  if($msg=~/$function/){
                        print "This is the msg: '$msg'\n";
                        my $send_msg="$SHIP_ID:".$SUBS{$function}->()."\n";
                        write_tty($send_msg);
                  }
                  else{
                        print "$SHIP_ID:Command not found!\n";
                  }
            }
      }
}
```

Listen function from vm.pl

```
sub write_tty{
      my $msg=\$_[0];
      print "writing to tty: '$msg'\n";
      open(TTY,">$TTY_PATH");
      flock(TTY, 2);       #Locks the tty for exclusive writing
      print TTY "$msg";
      flock(TTY, 8);       #Unlocks the tty
      close(TTY);
}
```

4.4 Testing

The testing performed for this project was both full feature and synthetic. The reason for performing both was that there were some parameters that had to be taken into account in order to make valid results. The main goal of the prototype is to see whether it is possible with the arcitechture designed to communicate two ways with running virtual machines in a cloud environment. Since the current environment does not support the creation of swarms of virtual machines simultainiously some of the tests need to be performed synthetically meaning partly simulated. This is to test the sustainability of the architecture suggested by this thesis.

4.4.1 Full feature testing

The real testing will create the foundation of the synthetic testing. The real tests were based on an experiment designed to collect the time used for 1 message to traverse from the machine running *send_msg.pl* script, to the queue, to the compute node running *compute.pl*, to a virtual machine over a tty, and all the way back to the machine running the *send_msg.pl* script.

Both the *send_msg.pl* and the *compute.pl* were rigged with timers in order to log the time usage the different parts of the process. The timer makes use of the module **Time::HiRes** in order to get the most accurate timestamps possible. In these tests these timers are denoted as *points of interest*. In total there were done 20 tests where each individual test sent 1 single message and the data were collected and analysed. Table 4.4.1 explains the different *points of interest* and the reason behind them.

After calculating the time usage of the different points of interest an analysis table combines the data and calcualtes the minimum, maximum, median, and average values of the tests. This table is denoted as 4.12. By reading the table a calculation was made on the average serial interaction time usage. This was done by subtracting the *"Time before writing to tty"* from the second time *"Done reading from tty"* appears. This is because this is the event in the script where it starts interacting with the serial interface. The result was an average serial interaction time of 61852,2857142857, which in seconds is approximately 0.62 seconds. This means in other words that the *"compute.pl"* script uses in average a total of 0.62 seconds when performing the necessary reading and writing operations to a tty.

The intention of the data collection tables 4.10, 4.11, and 4.12 was to collect time usage of the different spects of the *compute.pl* script. Most importantly was the time usage collected and denoted as serial interaction. Since it is only possible to emulate communication with large swarms of virtual machines, this time usage will be used by the *compute.pl* script when performing synthetic testing.

Point of interest	Description:
Started send_msg	This denotes the start of the send_msg tool before any messages have been sent.
Start callback, reading from queue	After the messages have reached the queue, the compute node will consume from it. This is done by calling the read_queue() function. The read_queue() function will call a callback function which will consume from the queue as long there are messages in it. This point of interest is timed before a message is consumed.
Time before writing to tty	After the message is consumed it is processed and written to the correct pts. Prior to the writing a timestamp is collected.
Lock aquired at	Before writing to the correct pts, a mutex is placed on the filehandle.
Lock released at	After the writing is performed, the lock is released.
Time after writing to tty	After the writing is completed and the lock is released, a timestamp is collected.
Time before reading from tty	Since the compute node will read the message it wrote for the virtual machine, the process is performed twice. The first read is done here.
Done reading from tty	The time is collected after the compute node is done reading from the corrected pts.
Time before reading from tty	This read collects the answer from the virtual machine, and a timestamp is collected prior the read_tty() function is called.
Done reading from tty	After reading the pts, another timestamp is collected in order to estimate the time used reading the pts.
Finished writing to queue	After the message has been read from the pts, it published back into the correct response queue, then a timestamp is collected.
Ended send_msg	The send_msg tool running on the machine that initiated the communication consumes the responses and ends the execution. A timestamp is collected when send_msg is done.

The data from table 4.10 was later on placed in an analysis table which calculated the difference in time usage based on these *points of interest*. This calculation will end in the total time usage of the round trip of the message from the machine running the *send_msg.pl* to a virtual machine running *vm.pl* and back. Below is an extract of the analysis table:

POI	Timestamps test 1	Timestamps test 2	Timestamps test 3
Started send_msg	139929560966900	139929687201028	139929694086607
Start callback, reading from queue	139929560968479	139929687201230	139929694087091
Time before writing to tty	139929560980875	139929687212762	139929694098497
Lock aquired at	139929560980886	139929687212775	139929694098508
Lock released at	139929560980890	139929687212779	139929694098512
Time after writing to tty	139929560980893	139929687212782	139929694098515
Time before reading from tty	139929560980884	139929687212770	139929694098504
Done reading from tty	139929560984729	139929687284622	139929694184640
Time before reading from tty	139929560984731	139929687284625	139929694184642
Done reading from tty	139929560995926	139929687295474	139929694195976
Finished writing to queue	139929560995999	139929687295549	139929694196043
Ended send_msg	139929561394844	139929687628608	139929694514168

Table 4.10: Raw data results from three tests collecting *points of interest*

Points of interest	Test 1	Test 2	Test 3
Started send_msg	0	0	0
Start callback, reading from queue	1579	202	484
Time before writing to tty	12396	11532	11406
Lock aquired at	11	13	11
Lock released at	4	4	4
Time after writing to tty	3	3	3
Time before reading from tty	-9	-12	-11
Done reading from tty	3845	71852	86136
Time before reading from tty	2	3	2
Done reading from tty	11195	10849	11334
Finished writing to queue	73	75	67
Ended send_msg	398845	333059	318125
Total time usage:	427944	427580	427561

Table 4.11: Results from calculating the difference in timestamps

Table 4.11 is an extract of a table which calculates the timestamps to $\frac{1}{100000}$ of a second. This was done for all 20 tests, and gives an estimate of the time usage of the different points of interest. *Started send_msg* denotes the time the tool was executed and a message was sent. The *points of interest* logs the time used for the important events run on the compute service. The previous *points of interest* are then deducted from the next *points of interest* in order to determine the time usage of every event. The reason for the negative values that appear at *Time before reading from tty* is that the reading has begun by a different process (the one that listens at the tty) prior to the last timestamp.

POI	Min	Max	Avg	Median
Time before writing to tty	-4377	1579	-1628,857	-990,5
Lock aquired at	10798	12431	11439,238	11334
Lock released at	10	20	11,952	11
Time after writing to tty	3	5	4,095	4
Time before reading from tty	2	3	2,381	2,5
Done reading from tty	-13	-9	-10,381	-10
Time before reading from tty	3845	86136	50917,000	51637,5
Done reading from tty	0	5	2,381	2
Finished writing to queue	7439	11845	10924,857	10988
Ended send_msg	65	77	70,286	69,5
Total time usage:	318125	398845	355916,381	355026
Time before writing to tty	427108	428068	427649,333	427711,5

Table 4.12: Analysis based on the results in table

4.4.2 Synthetic testing

Synthetic testing for this project is done by emulating writing to the tty's by the *compute.p* running on the compute nodes. In order to do this, multiple tests were run in advance in order to find the time the compute node uses to read and write to and from a tty. The synthetic experiments were conducted by placing a sleeping time of 0.62 seconds when the *compute.pl* script were intended to interact with a tty. This value was based on the average serial interaction time from the real tests.

The synthetic tests were first performed against a single compute node by defining n as the number of messages being sent and looping the *send_msg.pl* script 10 times with sleep time of s seconds. For every loop, the script would send n number of messages to the RabbitMQ server, retrieve the responses from the return queue, and sleep s seconds. The number of seconds of waiting time varied depending on how large n was. The intention of the sleep time was for the compute node to recover after retrieving n number of messages. The idea was to see what effect the arcitechture had on the resource usage of a compute node.

Example of looping send_msg.pl for synthetic testing

```
1 n=2000; for tall in $(seq 1 10 ); do echo "experiment $tall"; ./chief_send_msg.pl -m "command20"
2 -t /auto68.serialtest/ -w 150 -n $n
3 -f -q; sleep 30; done >
4 experiments/experiment_${n}_$(date +day_%m_time_%H_%M).dat
```

The command above was used to perform synthetic testing. There were performed four major experiments using synthetic tests. The experiments were conducted as follows:

- Tests towards a single busy compute node
  - 10 tests with 100 messages sent.

  - 10 tests with 500 messages sent.
  - 10 tests with 1000 messages sent.
  - 10 tests with 2000 messages sent.
- Tests towards a single non busy compute node
  - 10 tests with 100 messages sent.
  - 10 tests with 500 messages sent.
  - 10 tests with 1000 messages sent.
  - 10 tests with 2000 messages sent.
- Tests towards two busy compute nodes
  - 10 tests with a total of 100 messages sent.
  - 10 tests with a total of 500 messages sent.
  - 10 tests with a total of 1000 messages sent.
  - 10 tests with a total of 2000 messages sent.
- Test towards nine compute nodes
  - 10 tests with a total of 4500 messages sent

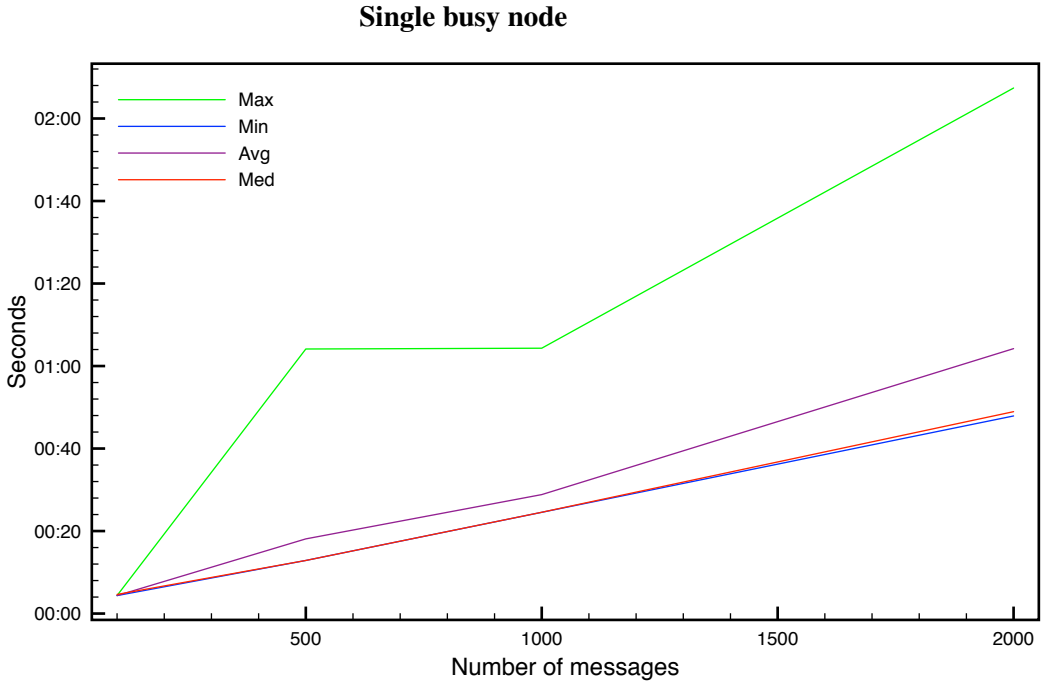

Figure 4.12: Average, median, maximum, and minimum values based on messages/time to a single compute node.

Graph 4.12 shows the plot of minimum, maximum, average and median values of the 10 tests conducted with 100, 500, 1000, and 2000 messages. This equals in total 40 tests for this experiment.

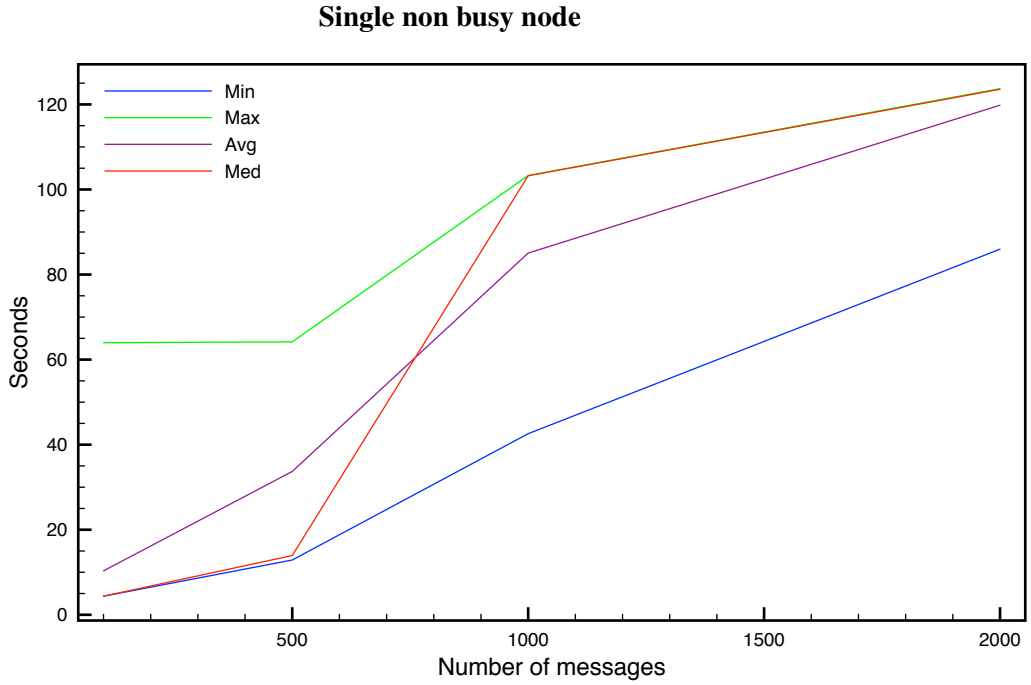

Figure 4.13: Average, median, maximum, and minimum values based on messages/time to a single compute node with low activity.

The graph 4.13 shows the results of the exact same tests as 4.12 except that they were performed against a compute node which had a significantly fewer running virtual machines.

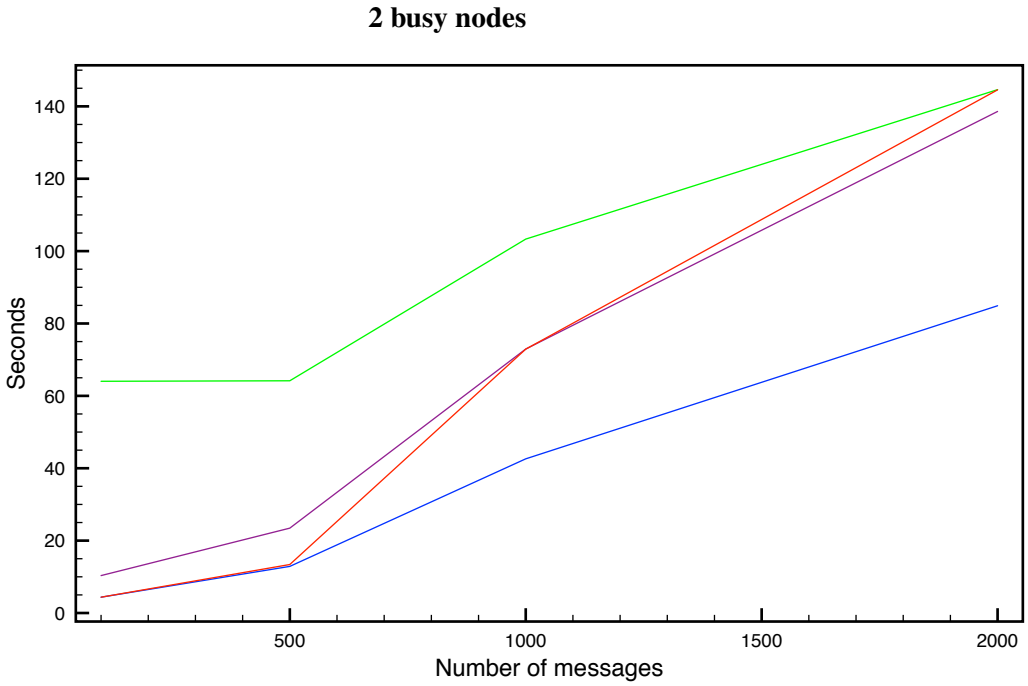

Figure 4.14: Average, median, maximum, and minimum values based on messages/time to 2 compute nodes.

Graph 4.14 also shows the minimum, maximum, average, and median values of a total of 40 tests. Opposed to the two previos graphs which shows values of ecperiments towards a single compute node, this graph shows values towards two busy compute nodes.

The last and final experiment that was performed towards 9 compute nodes to test The intention of this experiment was to test the architecture on a bigger scale and push the implementation to the maximum. The experiment was divided into three parts:

- 10 messages to each compute node, a total of 90 each time.
- 100 messages to each compute node, a total of 900 each time.
- 500 messages to each compute node, a total of 4500 each time.

The three tests described in the list above where all tests performed 10 times, the same way as the previous tests with a sleep time between each test performed. Before testing the architecture to the maximum, the first experiment sends 10 messages to each compute node, a total of 90 messages per test. 90 messages in 10 tests makes 900 messages in total for all tests.

Test number:	Messages sent:	Messages received:	Total time usage:
1	90	90	3,33
2	90	90	3,33
3	90	90	3,34
4	90	90	4,38
5	90	90	3,34
6	90	90	3,34
7	90	90	3,33
8	90	90	3,34
9	90	90	3,34
10	90	90	3,33
Total:	900	900	34,39

Table 4.13: 90 messages to 9 nodes 10 times, a total of 900 messages

Table 4.13 shows the results after sending 10 messages to 9 compute nodes 10 times. As the table shows, all messages were responded to and the round trip of 900 messages was 34, 39 seconds. This implies 26,17 responses on average per second and response success rate of 100%.

Test number:	Messages sent:	Messages received:	Total time usage:
1	900	900	22,51
2	900	899	22,49
3	900	900	21,48
4	900	900	22,52
5	900	899	22,48
6	900	900	22,45
7	900	900	22,55
8	900	898	22,51
9	900	900	22,49
10	900	900	22,52
Total:	9000	8996	224,00

Table 4.14: 100 messages to 9 nodes 10 times, a total of 9000 messages

The second part of the experiment sent 100 messages to 9 compute nodes 10 times which makes a total of 9000 messages sent out. Data collected for the experiment is shown in table 4.14. Of the 9000 messages sent out to 9 compute nodes, 8996 where responded to. This gives a response success rate of 96,32% and an average response time of 38,7 messeages per second.

Test number:	Messages sent:	Messages received:	Total time usage:
1	4500	3653	56,72
2	4500	3753	56,59
3	4500	2636	56,98
4	4500	2313	59,84
5	4500	2444	58,56
6	4500	2450	58,42
7	4500	3823	58,37
8	4500	2299	58,51
9	4500	2795	57,95
10	4500	2151	55,12
Total:	45000	28317	577,06

Table 4.15: 500 messages to 9 nodes 10 times, a total of 45000 messages

The last and final data collected is shown in table 4.15. This is the largest amount of messages tested against the architecture as a whole. 500 messages were sent to 9 compute nodes 10 times which makes a total of 45000 outgoing messages. Of the total of 45000 messages going out, 28317 responses came back. This implies a response success reate of 62,92% and an average of 49,07 messages per second.

Chapter 5

Analysis

This chapter aims to provide analysis of the data and findings of the result chapter and provide the reader with an analytical perspective of the overall project. Following this chapter is the discussion chapter which will discuss different aspects of how the project has been from start to end.

As the approach and the result chapter as are divided into different phases, these phases will be analysed seperately in this chapter. In addition, the overall outcome of the project will be analysed as a whole.

5.1 Analysis of the exploration phase

5.1.1 Virtual machines and choice of operating system

This project is based on the work done with the micro virtual machines [15]. The micro virtual machines in the mentioned project were run without any operating system and were as small as 512 bytes per instance. The intention of this project was whether or not if it was possible to send messages to swarms of virtual machines over virtual serial interfaces. Since this is an exploratory thesis the main goal has been all along to attempt to prove a potential concept of communicating over the tty's between the virtual machines and their hypervisors. Since it is a prove of concept, an operating system such as Ubuntu 12.04 was chosen for the virtual machines in this project. This was found to be a suitable choice for attempting to prove the concept at hand. Ubuntu is a complete operating system with support of different programming languages and modules. Even if its size is huge compared to the micro virtual machines, it is well suited for this project.

5.1.2 Tty's and serial interfaces

The intention of the exploration phase was to investigate how tty's work between a hypervisor and a virtual machine running on the cloud service Openstack. An interesting aspect with Openstack was that it was already making use of the tty's for logging to a console in the web interface Horizon. With this in mind, the exploration of the use of ttys was even more exciting. By noticing that Openstack already had implemented a way to utilize the use of tty's may be an indication that this concept has some potential.

The approach proposed in this thesis wanted to make use of the tty's as two way communication channel between a hypervisor and a virtual machine. The projects cloud implementation environment used KVM/Qemu as the virtualization technology. This ment that in addition to examining how data was read and written to and from the tty's in Linux as an operating system, it was also need for an examination on how this was effected when run inside a Qemu virtual machine.

It was discovered that the serial interface communication going from the hypervisor to the virtual machine actually is going through two kernels which increases the complexity in the way that there now exists two master-slave pairs towards a pts instead of just one. A normal master-slave relationship denotes the master side as the process which initiates the serial device, and the slave side as independent processes or devices which hooks up to the serial device. But in this case there exists a master-slave relationship on the compute node working as the hypervisor, where the master side is owned by Qemu and the *compute* service is the slave. At the same time, the master side is mapped to a serial interface on a virtual machine, which itself has a master-slave pair. This is an interesting find which needs more research in order to cover what impact this has on the proposed architecture of this thesis.

5.2 The tool and services

As the intention of this thesis has been to prove the concept of communicating with swarms of virtual machines it was vital that the tool and services developed for the prototype performed with the intended functionality.

The *send_msg* tool performed as intended and expected. The integrated functionality of the tool was proven to satifactory in the process of showing the functionality of the prototype.

The *compute* service on the other hand experienced som issues with freezing when receiving large amounts of traffic. In addition it was also first tested out using threading which did not turned out be a success. When implemented with threading, there were situations where service would block itself when trying to read from the tty. This may have something to do with the mutex not performing the way it was desired. There may have been some conflicts withing the threads when acquiring and releasing the locks on the tty's for reading and writiting.

Since threading caused problems proving the functionality of the prototype it was decided to switch to another type of parallelization by calling a script to handle incoming messages within the service itself. That way, the *compute* service could be listening and consuming messages from the queue, while consumed messages could be handled and written to the correct pts simultaneously. This created the possibility of parallelization of the message handling. Even though this particular way may not be the most effecient one, it was satisfying for the prototype.

The *vm* service running on the virtual machines was along with the *send_msg* tool proven to function according to expectations. It was able to handle incoming bytes on the serial device, read the string, and execute a dummy function based on the message. This was implemented to show how a virtual machine possibly could be asked to perform a certain task.

5.3 Analysis of test results

This section will look into the values collected through both the realistic and synthetic tests. The values will be analysed and evaluated concerning performance and scalability, and whether or not the tests were considered a success.

5.3.1 Realistic tests

The realistic tests were concidered a success because two-way serial communication was established through the architecture. It was proven through 20 subsequent tests, where a message was passed to the RabbitMQ server, collected by a compute node, and forwarded to a running virtual machine. The virtual machine was then able to respond to the message, and it was forwarded back to the *send_msg* tool through the queue.

The final results of the tests combined in table 4.12, gives an indication of what elements of the *compute* service that performing well. The time spent writing to a tty shows no cignificant difference or spread between its values. This implies that the writing process to a tty, is pretty predictable. The same goes for reading from the tty, even though there exist a slight spread between the minimum an maximum values. Even so, the median and average values are fairly close to one another. Writing to the queue has relatively high values compared other elements, which could be an indication of a potential scalability problem.

5.3.2 Synthetic tests

The graphs 4.12, 4.13, and 4.14 all show a spike when the number of messages exceeds 500. A possible reason for this spike may be exhaustion of file descriptors and tcp connections. Also, the fact that a tcp connection remains in use for several seconds after the connection is finished may contribute to the loss observed for the longer tests.

The graph 5.1 combines the values collected from the experiments conducted towards 9 compute nodes with the message amounts of 10, 100, and 500 message per node. There are not enough measurements to make a certain conclusion, but it appears likely that the time usage increases linearly. This is promising for the sustainability of the architecture. There was little variation in the results when the amount of messages was below 500. The same problem appeared again, as with the previous tests, when the message amount was 500, there was experienced delays and dropping of messages.

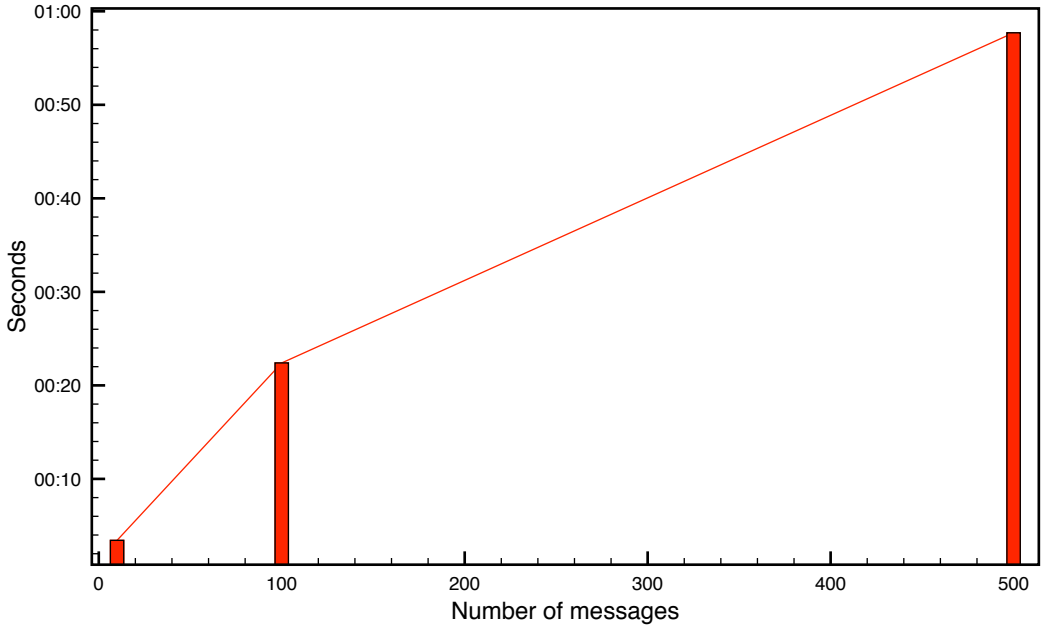

Figure 5.1: Average time of 10, 100, and 500 message to 9 compute nodes.

Chapter 6

Discussion

This chapter will cover and discuss results, prototype, testing and the process of this project as whole. It will suggest modifications to the approach this project has taken and discuss challenges met along the way. The chapter will also revisit the problem statement and conclude whether or not a goal has been reached.

6.1 Prototype as a framework

The possibility of making use of the prototype as a framework, or at least turning it into a framework is present. In retrospect a working tool(send_msg) and two services(compute service and vm service) have been developed, with a main focus on *send_msg* as a possible API(Application programming interface). A working prototype and a suggested architecture have been proposed in order to manage communication with micro virtual machines.

6.1.1 send_msg as a tool

send_msg has been suggested as possible API for future applications. The *send_msg* tool is, along with the *vm* service, the element of the architecture that has been proven to function best. *send_msg* has been implemented with different properties in order to support asynchronous communication with virtual machines running in a cloud environment. There exists several features that could have been implemented, but for the concept of proving asynchrounous communication over a virutal serial interface these have been sufficient. *send_msg* as a tool has performed satisfactory and can be applied for future work.

6.1.2 compute as a service

With regards to *compute* as a service for this communication prototype, it has performed sufficently for hundreds of incoming messages. When the number of messages increases some issues concerning scalabilty, speed of message handling, tty locking, and threading appears. In order for *compute* as a service to work sufficently enough in the proposed architecture, some improvement is needed. A proposed solution to improve theses aspects is to rewrite the service using a different programming language that has better support for threading and parallelization than perl has. A good idea would probably be to use a low level

language such as C or C++. Perl was chosen for this prototype because it provided the functionality needed to prove the concept of problem statement within the given time. It was concidered more important to prove the functionality of the prototype rather than focusing on how fast the message handling were performed. If the focus had been on making the prototype as optimal as possible, this may have resulted in not being able to prove the functionality

6.1.3 vm as a service

The service that has been run on the virtual machines during the testing of this prototype is the *vm* service. Along with *send_msg* it has been one of the two elements which has functioned satisfactory in the testing of the prototype. This service can also be applied for future work.

6.1.4 Architecture

The architecture described for this prototype incorporates the tool, the 2 services, as well as queueing system. The architecture achieved several important aspects stated early in the thesis. Among these aspects was asynchronous communication by the use of RabbitMQ. The use of a queueing system appeared to work for a certain amount of messages. During the testing it was shown that it may have had some impact on the prototype that the RabbitMQ server was running on a virtual machine. After moving the queueing system to a physical machine with better hardware specifications, this improved the performance of the prototype. In a cloud the virtual machines are running as software on top of a hypervisor which often is a physical machine. As perfomance becomes an important factor, a virtual machine in the cloud will not perform as a well as a designated physical server. So the environment SIREN will be set up in at this point, needs the proper hardware.

6.2 Serial communication

Probably one of the biggest obstacles in developing the prototype was fully comprehending how the tty's work when there exists two master-slave relationships over two kernels.. In addition, some of the problems which appeared throughout the testing may have been caused by the virtual machines own activity to and from tty's, as well as not being able to truly lock the tty devices. By own activity, this means the virtual machine writing to the tty devices on its own, not as a part of this prototype.

The concept gets more complex when leaving the normal master-slave relationship. Writing to and reading from a tty on a single kernel system i concidered pretty straight forward. The comlexity increases when writing and reading to and from a double set of tty's where one of the pair's master side is owned by Qemu. This is an element of the prototype which definitly needs more research. It was chosen not to pursue this any further in this thesis because the main focus was the functionality and testing of the prototype on a large scale, even if this ment exceeding the number of available virtual machines. With that being said, this is definitly

an aspect which should be prioritized when further development is done. This is proposed as a potential project for bachelor students.

6.3 Testing

The tests were carried out as planned even though some issues were experienced with the compute service freezing up, and some issues concering ttys. All the tests were conducted in a production environment which caused some challanges due to limitation of priviliges. In addition all experments needed to be planned in advance due to consideration on fellow students performing experiments in the same environment. Since it is a production environment, there never any certainty that the results produced from the tests are unaffected by others. So by using a live environment, some precautions were taken.
In the future, a tool should be developed in order to automatically test the architecture and perform an analysis on the performance.

6.4 Contributions to other research

This prototype could be possibly be with further development an asset or at least a be contribution to other projects which are working with cloud development or virtual machine development. Especially for development of custom virtual machines. As the micro virtual machines[15] at its current state only allows for serial device communication, this prototype could be an asset in sense of ordering wake up for virtual machines or allow the hypervisor to request performance status on how the virtual machines are doing. [20] discusses the use of particle swarm optimization algorithm. The architecture proposed in this thesis could be used as communication channel between virtual machines for the use of the algorthm.

6.5 Future work

The prototype as a framework has been designed to focused on large scale functionality rather than speed. For system administrators both aspects are important. In order to improve the prototype it is suggested rewriting it using a low level programming language in order to better support parallelization through threading, as well as increasing the overall speed of the functionality.

Since the prototype only was valid for one of the four communication types in the design, suggested work should include developing and implementing the three communication possibilities left out of this thesis. This would further create the opportunities of reaching the wanted functionalities such as:

- Allow for virtual machines to report on their current performance in order to establish perceived quality of service
- Implement the possibility for the VM to order "wake up" from sleeping from the hypervisor

- Use the *send_msg* tool to implement and utilize the possibility of create dynamic scheduling.
- Allow for virtual machines to question other virtual machines on their perceived quality of service in order to estimate what their own quality of service should be.
- Implement the possibility for virtual machines to request live migration.
- Use BSON as a format for interchanging messages in order to further minimize the data traversing the communication channel.

The prototype as it is currently is neither stabile enough or possesses the functionality to implement these future works at the given time. But it has shown the concept of communicating with virtual machine over a virtual serial device and proven that this is possible.

6.6 Problem statement: A lookback

6.6.1 Has the goal of the thesis been reached?

This thesis has achieved a valid answer to the problem statement decleared. Even though there are some elements that are not completed this does not imply that the suggested solution is invalid. The architecture and prototype presented is a piece of a puzzle which needs more research in order to perform at a production environment level. A concept has been proven throughout this project which is applicable to further investigate, which was the goal at the starting point.

Chapter 7

Conclusion

The goal of this master thesis has been to explore the possibility of managing the communication to swarms of micro virtual machines in a cloud environment. The thesis proposes a way of achieving this in order to help system administrators through a new management paradigm as cloud computing worldwide grows larger along with the need for energy consumption. The result was a proposed architecture which allows for communication with mulitple virtual machines over virtual serial interfaces, to explore and determine how this is possible in a cloud environment such as Openstack.

This thesis has paved way for communication with micro virtual machines, which is a necessity if the world going towards cloud environments that are depending on a reduction in power consumption. The possibility of using the virtual serial interfaces for communication with swarms has been proven to exist. The thesis has proposed an architecture which allows for communication with thousands of virtual machines. The design utilized a messaging queue which was able to scale through thousands of messages within seconds. Some stability concerns regarding message queue and serial communication as well as threading, have been uncovered and extensively discussed.

Even though the prototype itself has some shortcommings, the main goal was to prove a concept. What has been proposed is evaluated to be good enough to keep developing. This is with regards to the management aspect which now could make use of the protocol, pending a solution to the scalability issue.

Bibliography

[1] Ant colony optimization, wikipedia. http://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms.

[2] Bson website. http://bsonspec.org/.

[3] Kvm reignites type 1 vs. type 2 hypervisor debate. http://searchservervirtualization.techtarget.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate.

[4] Libvirt website. http://libvirt.org/index.html.

[5] Linux pty man page. http://man7.org/linux/man-pages/man7/pty.7.html.

[6] Openstack site, wikipedia. http://en.wikipedia.org/wiki/OpenStack.

[7] Openstack website. http://www.openstack.org/software.

[8] Particle swarm optimization, wikipedia. http://en.wikipedia.org/wiki/Particle_swarm_optimization.

[9] Rabbitmq website. https://www.rabbitmq.com/.

[10] The tty demystified. http://www.linusakesson.net/programming/tty/.

[11] Understanding amqp, the protocol used by rabbitmq. http://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/.

[12] Kvm - kernel-based virtual machine. http://www.redhat.com/rhecm/rest-rhecm/jcr/repository/collaboration/jcr:system/jcr:versionStorage/5e7884ed7f00000102c317385572f1b1/1/jcr:frozenNode/rh:pdfFile.pdf, 2009.

[13] How clean is your cloud. greenpeace climate reports. http://www.greenpeace.org/international/en/publications/Campaign-reports/Climate-Reports/How-Clean-is-Your-Cloud/, 2012.

[14] Predicting enterprise cloud computing growth. http://www.forbes.com/sites/louiscolumbus/2013/09/04/predicting-enterprise-cloud-computing-growth/, 2013.

[15] Alfred Bratterud and Hårek Haugerud. Maximizing hypervisor scalability using minimal virtual machines. Technical report, Oslo and Akershus University College of Applied Science, 2013.

[16] Eugen Feller, Christine Morin, and Armel Esnault. A case for fully decentralized dynamic vm consolidation in clouds. Technical report, 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, 2012.

[17] Lava Computer MFG inc. Rs-232: Seriel ports. http://lpvo.fe.uni-lj.si/fileadmin/files/Izobrazevanje/OME/rs_232_serial_ports.pdf, 2002.

[18] New York Times James Glanz. Google details, and defends, its use of electricity. http://www.nytimes.com/2011/09/09/technology/google-details-and-defends-its-use-of-electricity.html, 2011.

[19] PowerTCO Ken Milberg. Ibm and hp virtualization: A comparative study of unix virtualization on both platforms. https://www.ibm.com/developerworks/aix/library/au-aixhpvirtualization/, 2009.

[20] N. Nagaveni R. Jeyarani and R. Vasanth Ram. Self adaptive particle swarm optimization for efficient virtual machine provisioning in cloud. Technical report, International Journal of Intelligent Information Technologies, 2011.

[21] Maiko Shigeno Hidemoto Nakada Tomohiro Kudoh Satoshi Takahashi, Atsuko Takefusa and Akiko Yoshise. Virtual machine packing algorithms for lower power consumption. Technical report, 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, 2012.

[22] Kai Lu Yongpeng Liu, Hong Zhu and Xiaoping Wang. Self-adaptive management of the sleep depths of idle nodes in large scale systems to balance between energy consumption and response times. Technical report, 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, 2012.

Chapter 8

Apendices

8.1 send_msg.pl

send_msg.pl

```
1  #!/usr/bin/perl
2
3  # our needed packages
4
5  use strict "vars";
6  use Getopt::Std;
7  use DBI;
8  use Net::RabbitFoot;
9  use List::MoreUtils qw(uniq);
10 use AnyEvent;
11 use Time::HiRes;
12
13 # global variables
14
15 my $VERBOSE = 0;
16 my $DEBUG = 0;
17
18 # command line options
19
20 my $opt_string = "vdht:m:ifc:D:l:w:c:qn:C:";
21 getopts("$opt_string", \my %opt) or usage() and exit(1);
22
23 $VERBOSE = 1 if $opt{'v'};
24 $DEBUG = 1 if $opt{'d'};
25
26 my $CONN; #Global connectection to rabbitMQ
27 my $CHAN; #Channel for rabbitMQ connection
28
29 my $ID = 1 if $opt{'i'};
30 my $TARGET = $opt{'t'};
31 my $MSG = $opt{'m'};
32 my $WTIME= $opt{'w'};
33 my $BLOCK = $opt{'b'};
34 my $MAILBOX = $opt{'c'}; #User can check for messages in existing queue
35 my $QUIET= 1 if $opt{'q'};
36 my $SIM=1 if $opt{'f'}; #Simulates traffic
37 my $NUM_OF_MSGS=$opt{'n'};
38 my @sim_queues=("compute08");
39 @sim_queues = split /:/,$opt{'C'} if $opt{'C'};
40
41 print "NUM_OF_MSGS: $NUM_OF_MSGS\n";
42
43 my $EMPT_RES;
44
```

```
45  my $RES_COUNT=0; #Keep track of number of responses
46  my $MSG_COUNT=0; #Keep track of number of outgoing messages
47
48  my $arg_count=0; #Used for excluding arguments
49
50  my $SHIP_ID;
51  my $READ_QUEUE;
52  my $SHIP_MSG_ID;
53
54  my %OUTGOING;
55  my %INCOMING;
56
57  if($opt{'h'}){
58      usage();
59      exit 0;
60  }
61
62  if(!$MAILBOX){
63      usage() and die("You have to supply a message with -m <message>") unless $MSG;
64  }
65  $arg_count++ if($ID);
66  $arg_count++ if($WTIME);
67  $arg_count++ if($BLOCK);
68  $arg_count++ if($MAILBOX);
69
70  usage() and die("You can only supply one of these arguments simultainously: -i <id>, -w <wait>,
71  -b <block>, -c <queue ID>") if $arg_count>1;
72
73
74
75  ##################################################
76  # Main content
77
78  my $start = Time::HiRes::time;
79  print "TRACE: Started script at:" . $start . "\n";
80  print "TRACE: Start normaltime: " . time . "\n";
81
82  if($MAILBOX){
83      while(1){
84          check_mail();
85          if(!$EMPT_RES){
86              last;
87          }
88          $EMPT_RES=0;
89      }
90
91      print "Collected $RES_COUNT messages from $READ_QUEUE.\n";
92      exit 0;
93  }
94  else{
95      rmq_connect();
96      construct_outgoing();
97      print_current_outgoing();
98      create_response_queue();
99      if($SIM){
100         print "number of msgs being sent: ".$NUM_OF_MSGS."\n";
101
102         sim_create_outgoing_queues(@sim_queues);
103         my $counter=0; #Keep track of every 100 msg sent. Sleep in between
104         for(my $i=0;$i<=$#sim_queues;$i++){
105             for(my $j=0;$j<$NUM_OF_MSGS;$j++){
106                 sim_send_msg($sim_queues[$i]);
107                  if($counter==100){
108                      sleep 2;
109                      $counter=0;
110                  }
111                  $counter++;
112             }
```

```
113             }
114         }
115         else{
116             create_outgoing_queues();
117             send_msg();
118         }
119
120         if($ID){
121
122             print "Response information:\n";
123             print "\tShipment id: $SHIP_ID\n";
124             print "\tReturn queue: $READ_QUEUE\n";
125         }
126         elsif($WTIME){
127             my $counter=0;
128             my $interval=1;
129
130             while($counter<=$WTIME){
131                 #Checks is the number of responses matches the number of outgoing messages
132                 if($RES_COUNT<$MSG_COUNT){
133                     read_queue();
134                     sleep $interval;
135                     $counter+=$interval;
136                 }
137                 else{
138                     last;
139                 }
140             }
141         }
142
143         my $end=Time::HiRes::time;
144         print "TRACE:Finished script at HiRes:" . $end . "\n";
145         print "TRACE:EndTime normaltime" . time . "\n";
146         print "TRACE:Total time spent: " .($end - $start) . "\n";
147         print "Sent $MSG_COUNT messages.\n";
148         print "Recieved $RES_COUNT responses.\n";
149         print "close connection\n";
150         rmq_close();
151     }
152
153
154
155     ##################################################
156     # Subroutines
157
158
159     sub usage {
160
161         print "Usage:\n";
162         print "-h for help\n";
163         print "-v for verbose(more output)\n";
164         print "-d for debug(even more output)\n";
165         print "-t <target> Send message to vm(s) as target where target is mysql pattern\n";
166         print "-i Returns shipment ID and queue ID for shipment.\n";
167         print "-D <ID> delete queue from inbox with id: <ID>\n";
168         print "-l <ID> list messages in queue with id: <ID>\n";
169         print "-w <seconds> Wait for response from vm for <seconds> seconds\n";
170         print "-c <queue id> Apply queue ID to collect messages\n";
171         print "-q Quiet mode. Does not print anything to console.\n";
172         print "-f Fake mode. Simulation.\n";
173         print "-n <number of message to a VM>.\n";
174     }
175
176     sub verbose {
177         print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
178     }
179
180     sub debug {
```

```
181         print "DEBUG: " . $_[0] if ($DEBUG);
182     }
183
184     ##
185     #
186     #Checks if $TARGET is a regular expression
187      #
188     ##
189     sub pattern_check {
190         if($_[0]=~/^\/.+?\/$/){
191             $TARGET=~s/\///g;
192             return 1;
193         }
194         else{
195             return 0;
196         }
197     }
198
199
200     ##
201     #
202     #Constructs hash for outgoing messages
203     #
204     #Parameters: None
205     #
206     #Return values: None
207     #
208     ##
209     sub construct_outgoing {
210         my $user = "*";
211         my $passwd = "*";
212         my $host = "10.10.10.51";
213         my $db = "*";
214         my $table = "*";
215
216         my $dbh = DBI->connect("DBI:mysql:$db;host=$host;",$user, $passwd);
217
218         if($dbh){
219             debug("Connected to the database!\n");
220
221             my $sql;
222             if(pattern_check($TARGET)){
223                 $sql="SELECT hostname, uuid, host FROM $table WHERE vm_state='active' and
224                  hostname LIKE '$TARGET'";
225             }
226             else{
227                 $sql="SELECT hostname, uuid, host FROM $table WHERE vm_state='active'";
228             }
229
230             my $query = $dbh->prepare($sql);
231             $query->execute();
232
233             $SHIP_ID = time;
234             $SHIP_ID*=int(rand(10000));
235             $SHIP_ID=~s/0/1/g;
236             if($SHIP_ID=~/(\d{7})(\d+)/){
237                 $READ_QUEUE=$1;
238                 $SHIP_MSG_ID=$2;
239             }
240
241             #Will not construct hash if the program runs as a simulation
242             if(!$SIM){
243                 if($query->rows){
244                     while(my @results = $query->fetchrow_array()){
245                         $OUTGOING{$results[0]}{$results[1]}{$results[2]}
246                         {$SHIP_MSG_ID}=$MSG;
247                     }
248                 }
```

```
249                     else{
250                         print "There were no results matching the query\n";
251                     }
252                 }
253             }
254     }
255
256     #
257     #Prints current datastructure
258     #
259     sub print_current_outgoing{
260         foreach my $vm (sort(keys %OUTGOING)) {
261             print "$vm\n\t";
262             foreach my $uuid (keys %{$OUTGOING{$vm}}){
263                 print "UUID: $uuid\n\t";
264                 foreach my $compute_node (keys %{$OUTGOING{$vm}{$uuid}}){
265                     print "Located at PM: $compute_node\n\t";
266                     foreach my $ship_msg_id (keys %{$OUTGOING{$vm}{$uuid}
267                     {$compute_node}}){
268                         print "Shipment_ID: $ship_msg_id\n\t";
269                         print "Content: $uuid:$READ_QUEUE:$ship_msg_id:$OUTGOING
270                         {$vm}{$uuid}{$compute_node}{$ship_msg_id}\n\n";
271                     }
272                 }
273             }
274         }
275     }
276
277
278     #
279     #Connects to the rabbitMQ server
280     #
281     sub rmq_connect{
282         $CONN = Net::RabbitFoot->new()->load_xml_spec()->connect(
283             host  => '10.10.10.51',
284             port => 5672,
285             user => '*',
286             pass => '*',
287             vhost => '/',
288         );
289     }
290
291     #
292     #Creates queues fro outgoing messages
293     #
294     sub create_outgoing_queues{
295         $CHAN = $CONN->open_channel();
296         my @cnodes;
297
298         foreach my $vm (sort(keys %OUTGOING)) {
299             foreach my $uuid (keys %{$OUTGOING{$vm}}){
300                 foreach my $compute_node (sort keys %{$OUTGOING{$vm}{$uuid}}){
301                     push (@cnodes, $compute_node);
302                 }
303             }
304         }
305
306         @cnodes = uniq @cnodes;
307
308         for(my $i=0; $i<=$#cnodes; $i++){
309             $CHAN->declare_queue(
310                 queue => $cnodes[$i],
311                 durable => 1,
312             );
313         }
314     }
315
316     #Create outgoing queues for simulation
```

```
317 sub sim_create_outgoing_queues{
318
319         my @cnodes=@_;
320
321         for(my $i=0; $i<=$#cnodes; $i++){
322                 $CHAN->declare_queue(
323                         queue => $cnodes[$i],
324                         durable => 1,
325                 );
326         }
327 }
328
329 #
330 #Creates a response queue
331 #
332 sub create_response_queue{
333         $CHAN = $CONN->open_channel();
334
335         $CHAN->declare_queue(
336                 queue => $READ_QUEUE,
337                 durable => 1,
338         );
339 }
340
341 #
342 #Send message to rabbitmq
343 #
344 sub send_msg{
345
346         foreach my $vm (sort(keys %OUTGOING)) {
347                 foreach my $uuid (keys %{$OUTGOING{$vm}}){
348                         foreach my $compute_node (keys %{$OUTGOING{$vm}{$uuid}}){
349                                 foreach my $ship_msg_id (keys %{$OUTGOING{$vm}{$uuid}
350                                 {$compute_node}}){
351                                         my $string = $OUTGOING{$vm}{$uuid}{$compute_node}{$ship_msg_id};
352                                         my $send_msg="$uuid:$READ_QUEUE:$SHIP_MSG_ID:$string";
353                                         $CHAN->publish(
354                                                 exchange => '',
355                                                 routing_key => $compute_node,
356                                                  body => $send_msg,
357                                         );
358                                         $MSG_COUNT++;
359                                 }
360                         }
361                 }
362         }
363 }
364
365 sub sim_send_msg{
366         my $sim_msg="e606cfc4-3900-4cd9-b429-d47dae526179:$READ_QUEUE:$SHIP_ID:$MSG";
367         my $compute_node=$_[0];
368
369         $CHAN->publish(
370                 exchange => '',
371                 routing_key => $compute_node,
372                  body => $sim_msg,
373         );
374         $MSG_COUNT++;
375 }
376
377
378 #
379 #
380 #
381 sub check_mail{
382         $READ_QUEUE=$MAILBOX;
383         read_queue();
384 }
```

```
385
386
387   #
388   #Reads from the current shipment queue
389   #
390   sub read_queue{
391         $CONN = Net::RabbitFoot->new()->load_xml_spec()->connect(
392               host => '10.10.10.51',
393               port => 5672,
394               user => '*',
395               pass => '*',
396               vhost => '/',
397          );
398
399         $CHAN = $CONN->open_channel();
400
401          $CHAN->declare_queue(
402                queue => $READ_QUEUE,
403                durable => 1,
404          );
405
406          if(!$QUIET){
407                print " [*] Waiting for messages. To exit press CTRL-C\n";
408          }
409          $CHAN->qos(prefetch_count => 1,);
410
411         $CHAN->consume(
412                on_consume => \&callback,
413                no_ack => 0,
414         );
415         if($MAILBOX){
416               AnyEvent->condvar->recv;
417         }
418   }
419
420   sub callback {
421         my $var = shift;
422            my $body = $var->{body}->{payload};
423         $body=~s/#/0/g;
424         if(!$QUIET){
425               print " [x] Received $body\n";
426         }
427         ##
428            my @c = $body =~ /\./g;
429            sleep(scalar(@c));
430         if(!$QUIET){
431                print " [x] Done\n";
432            }
433         $CHAN->ack();
434         $RES_COUNT++;
435         $EMPT_RES=1;
436   }
437
438   #
439   #Closes the rabbitMQ connection
440   #
441   sub rmq_close{
442         $CONN-close();
443   }
```

8.2 compute.pl

compute.pl

```
#!/usr/bin/perl

# Our needed packages

use strict "vars";
use strict "refs";
use Getopt::Std;
use threads;
use Time::HiRes;
use Net::RabbitFoot;
use Time::HiRes qw(usleep);use List::MoreUtils qw(uniq);

# global variables

my $VERBOSE = 0;
my $DEBUG = 0;


# command line options
my $opt_string = "vdht:m:i:bc:D:l:w:fA:";
getopts("$opt_string", \my %opt) or usage() and exit(1);

$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};
my $TTY_PATH;       #pts path to VM
my $READ_QUEUE="compute02"; #Queue for incoming traffic
my $WRITE_QUEUE; #Queue for outgoing traffic
my $CONN; #Global connectection to rabbitMQ
my $CHAN; #Channel for rabbitMQ connection
my $ANYEVENT;
my $MSG_SENT = 0;
my $VM_UUID; #This is the UUID for the vm to recieve traffic. This is intended for collecting the
correct tty channel
my $VM_MSG; #Message which is forwarded to vm
my $VM_RESPONSE;        #Response which is sent from VM based on the previous message it
recieved
my $SHIP_ID; #Current shipment ID for message
my $SIM=1 if $opt{'f'};
my $ANSWER = $opt{'A'};


my @threads; #Stores all threads in an array
my $t; #thread

my $interval=1;
my $EMPT_RES;

if($opt{'h'}){
        usage();
        exit 0;
}


if ( $ANSWER ){

    # we have been called to answer a request.
    handle_msg($ANSWER);

    exit 0;
}


##################################################
# Main content
my $start=Time::HiRes::time;
read_queue();
```

```
67  rmq_close();
68
69  for(my $i=0;$i<$#threads;$i++) {
70          $threads[$i]->join;
71  }
72
73
74  ##################################################
75  # Subroutines
76
77
78  sub usage {
79
80          print "Usage:\n";
81          print "-h for help\n";
82          print "-v for verbose(more output)\n";
83          print "-d for debug(even more output)\n";
84          print "-b \n";
85          print "-t <target> Send message to vm(s) as target where target is mysql pattern\n";
86          print "-i <queue ID> apply queue ID\n";
87          print "-D <ID> delete queue from inbox with id: <ID>\n";
88          print "-l <ID> list mesages in queue with id: <ID>\n";
89          print "-w <seconds> Wait for response from vm for <seconds> seconds\n";
90  }
91
92  sub verbose {
93          print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
94  }
95
96  sub debug {
97          print "DEBUG: " . $_[0] if ($DEBUG);
98  }
99
100 sub listen_tty{
101     my $tobject = threads->self;
102     my $tid = $tobject->tid;
103     print "Listen is tid $tid\n";
104     if($SIM){
105         print "Sleeping 0.62 seconds\n";
106         usleep(61852);
107         write_to_queue("e606cfc4-3900-4cd9-b429-d47dae526179:Fake Return");
108         print "Finished writing message to queue\n";
109     }
110     else{
111         my $temp = new_read_tty();
112         chomp($temp);
113         print "$tid: This is temp: '$temp'\n";
114         my $msg=new_read_tty();
115         chomp($msg);
116         print "$tid: This is the response: '$msg'\n";
117         if($msg=~/\d+:.+/){
118             print "$tid: Writing current message to queue: '$msg'\n";
119             write_to_queue($VM_UUID.$msg);
120             my $end=Time::HiRes::time;
121             print "TRACE:$VM_UUID:Finished writing to queue:". $end . "\n";
122                     print "TRACE:$VM_UUID:Time spent processing 1 msg:" . ($end-$start) . "\n";
123         }
124     }
125 }
126
127 sub write_tty{
128         my $write_tty_start = Time::HiRes::time;
129         print "TRACE:$VM_UUID:Time before writing to tty:" . $write_tty_start . "\n";
130         my $msg=$_[0]."\n";
131         print "writing to tty: '$msg'";
132         open(TTY,">$TTY_PATH");
133         print "Aquiring lock\n";
134         flock(TTY, 2); #Locks the tty for writing
```

```
135         my $start_lock_time = Time::HiRes::time;
136         print "TRACE:$VM_UUID:Lock aquired at:$start_lock_time \n";
137         print TTY $msg;
138         print "Releasing lock\n";
139         flock(TTY, 8); #Unlocks the tty
140         my $end_lock_time=Time::HiRes::time;
141         print "TRACE:$VM_UUID:Lock released at:$end_lock_time \n";
142         close(TTY);
143         my $write_tty_end=Time::HiRes::time;
144         print "TRACE:$VM_UUID:Time after writing to tty:" . $write_tty_end . "\n";
145     }
146
147     sub read_tty{
148         open(TTY, "$TTY_PATH");
149         my $line = <TTY>;
150         chomp($line);
151         print "LINE $line\n";
152         my @content=split(":",$line);       #splits message
153         if($content[0]=~/$SHIP_ID/){
154             $VM_RESPONSE=$content[1];
155             return $VM_RESPONSE;
156         }
157         close(TTY);
158         return $line;
159     }
160
161     sub read_all{
162         open(TTY, "$TTY_PATH");
163         while(my $line = <TTY>){
164             print "$line";
165         }
166         close(TTY);
167     }
168
169     sub test_thread{
170         my $num=shift;
171         print "Thread $num started\n";
172         sleep 3;
173         print "done with thread number $num\n";
174     }
175
176     #
177     #Collects the correct pts based on the VMs UUID
178     #
179     sub get_tty{
180         if($SIM){
181             open(VIRSH,"virsh dumpxml e606cfc4-3900-4cd9-b429-d47dae526179 |");
182         }
183         else{
184             open(VIRSH,"virsh dumpxml $VM_UUID |");
185         }
186         while(my $line = <VIRSH>){
187             if($line=~/<serial type=\'pty\'>/){
188                 $line=<VIRSH>;
189                 if($line=~/<source path='(\/dev\/pts\/\d+)'\/>/){
190                     $TTY_PATH=$1;
191
192                     last;
193                 }
194             }
195         }
196         close(VIRSH);
197     }
198
199     #
200     #Sets necessary global variables and paths based on the incoming message
201     #Parameter: message body from RabbitMQ message
202     #
```

```
203 sub construct{
204         my $body = $_[0]; #message from RabbitMQ
205         my @content=split(":",$body);        #splits message
206         $WRITE_QUEUE=$content[1];        #Stores correct return queue
207         $VM_UUID=$content[0];        #Stores UUID for forwarding message through serial interface
208         $SHIP_ID=$content[2];
209         $VM_MSG="$SHIP_ID:$content[3]";
210         get_tty();        #Collects correct pts interface
211 }
212
213 #RABBITMQ
214
215 #
216 #Reads from queue specified in for this Compute node
217 #
218 sub read_queue{
219         $CONN = Net::RabbitFoot->new()->load_xml_spec()->connect(
220                 host => '10.10.10.51',
221                 port => 5672,
222                 user => '*',
223                 pass => '*',
224                 vhost => '/',
225         );
226
227         $CHAN = $CONN->open_channel();
228
229         $CHAN->declare_queue(
230                 queue => $READ_QUEUE,
231                 durable => 1,
232         );
233
234         print " [*] Waiting for messages. To exit press CTRL-C\n";
235         $CHAN->qos(prefetch_count => 1,);
236
237         $CHAN->consume(
238                 on_consume => \&callback,
239                 no_ack => 0,
240         );
241
242         AnyEvent->condvar->recv;
243 }
244
245 #
246 #
247 #
248 sub write_to_queue{
249     my $msg = $_[0];
250     print "Attempting to return message: '$msg'\n";
251     print "Establishing conenction to write queue: $WRITE_QUEUE\n";
252     my $conn = Net::RabbitFoot->new()->load_xml_spec()->connect(
253         host => "10.10.10.51",
254         port => 5672,
255         user => '*',
256         pass => '*',
257         vhost => '/',
258     );
259
260     print "connection established\n";
261
262
263     $CHAN = $conn->open_channel();
264     $CHAN->declare_queue(
265         queue => $WRITE_QUEUE,
266         durable => 1,
267     );
268
269     $CHAN->publish(
270         exchange => '',
```

```
271         routing_key => $WRITE_QUEUE,
272             body => $msg,
273         ) or warn("Could not print message...\n");
274     }
275     sub callback {
276         print "TRACE:$VM_UUID:Start callback, reading from queue:" . Time::HiRes::time . "\n";
277         my $var = shift;
278             my $body = $var->{body}->{payload};
279                 print " [x] Received $body\n";
280
281         system("/root/kjetil_master/return_answer.pl '$body' &");
282
283         my @c = $body =~ /\./g;
284         sleep(scalar(@c));
285
286         print " [x] Done\n";
287
288         print "TRACE:$VM_UUID:Finished reading msg from queue:" . Time::HiRes::time . "\n";
289         $EMPT_RES=1;
290         $CHAN->ack();
291     }
292
293     sub handle_msg{
294         print "Handle message called\n";
295
296         my $body=$_[0];
297         construct($body);
298         $t = threads->new(\&listen_tty);
299         print "Tread constructed, sleeping\n";
300
301         sleep 3;
302         if(!$SIM){
303             print "Using tty: $TTY_PATH\n";
304             print "This is the correct return queue: $WRITE_QUEUE\n";
305             print "This is the correct return shipment ID: $SHIP_ID\n";
306             print "this is vms_msg:$VM_MSG\n";
307             write_tty($VM_MSG);
308         }
309
310
311         print "joining threads\n";
312         $t->join;
313
314     }
315
316     #
317     #Closes the rabbitMQ connection
318     #
319     sub rmq_close{
320         $CONN-close();
321     }
322
323
324     sub new_read_tty {
325         my $read_tty_start=Time::HiRes::time;
326         print "TRACE:$VM_UUID:Time before reading from tty:" . $read_tty_start . "\n";
327         use Term::ReadKey;
328
329         open(TTY, "<$TTY_PATH");
330         print "Gimme a char: ";
331         ReadMode "raw";
332         my $string;
333         while ( my $key = ReadKey 0, *TTY ){
334                 $string .= $key;
335         }
336         ReadMode "normal";
337         close(TTY);
338         print "Got string: '$string'\n";
```

```
339     my $read_tty_end=Time::HiRes::time;
340     print "TRACE:$VM_UUID:Done reading from tty:" . $read_tty_end ."\n";
341     chomp($string);
342         return $string;
343 }
```

8.3 vm.pl

vm.pl

```
1   #!/usr/bin/perl
2
3   # Our needed packages
4
5   use strict "vars";
6   use Getopt::Std;
7   use threads;
8   use Term::ReadKey;
9   use Sys::Hostname;
10  # global variables
11
12  my $VERBOSE = 0;
13  my $DEBUG = 0;
14
15  # command line options
16
17  my $opt_string = "vdht:m:i:bc:D:l:w:";
18  getopts("$opt_string", \my %opt) or usage() and exit(1);
19
20  $VERBOSE = 1 if $opt{'v'};
21  $DEBUG = 1 if $opt{'d'};
22  my $TTY_PATH="/dev/ttyS1";
23  my $VM_MSG;             #Message incoming to VM
24  my $SHIP_ID;       #Shipment ID for message
25  my $HOSTNAME = hostname;
26  $HOSTNAME=~s/0/#/g;
27
28  my %SUBS = (
29              "command1" => "action1",
30              "command2" => "action2",
31          );
32
33  if($opt{'h'}){
34          usage();
35          exit 0;
36  }
37
38
39  #usage() and die("You have to supply a message with -m <message>") unless $MSG;
40
41  ##################################################
42  # Main content
43
44  # empty tty (/dev/ttyS1)
45
46  my $t = threads->new(\&listen_tty);
47  # write_tty("command1");
48  $t->join;
49
50  #listen_tty();
51
52  ##################################################
53  # Subroutines
54
55
56  sub usage {
```

```
57
58          print "Usage:\n";
59          print "-h for help\n";
60          print "-v for verbose(more output)\n";
61          print "-d for debug(even more output)\n";
62      }
63
64      sub verbose {
65          print "VERBOSE: " . $_[0] if ($VERBOSE or $DEBUG);
66      }
67
68      sub debug {
69          print "DEBUG: " . $_[0] if ($DEBUG);
70      }
71
72      sub write_tty{
73          my $msg=$_[0];
74          print "writing to tty: '$msg'\n";
75          open(TTY,">$TTY_PATH");
76          flock(TTY, 2);      #Locks the tty for exclusive writing
77          print TTY "$msg";
78          flock(TTY, 8);      #Unlocks the tty
79          close(TTY);
80      }
81
82      sub listen_tty{
83          while(1){
84              my $msg=read_new_tty();
85              chomp($msg);
86              foreach my $function (keys %SUBS){
87                  if($msg=~/$function/){
88                      print "This is the msg: '$msg'\n";
89                      my $send_msg="$SHIP_ID:".$SUBS{$function}->()."\n";
90                      write_tty($send_msg);
91                  }
92                  else{
93                      print "$SHIP_ID:Command not found!\n";
94                  }
95              }
96          }
97      }
98
99      sub read_tty{
100         open(TTY, "$TTY_PATH");
101         my $line = <TTY>;
102         chomp($line);
103         my @content=split(":",$line);       #splits message
104         close(TTY);
105         $SHIP_ID="$content[0]";
106         $VM_MSG="$content[1]";
107         print "VM_MSG is: '$VM_MSG'\n";
108         return $VM_MSG;
109     }
110
111     sub read_all{
112         open(TTY, "$TTY_PATH");
113         while(my $line = <TTY>){
114             print "$line";
115         }
116         close(TTY);
117     }
118
119     sub action1{
120         my $action="$HOSTNAME returning action10";
121         return $action;
122     }
123
124     sub action2{
```

```
125         my $action="$HOSTNAME returning action20";
126         return $action;
127     }
128
129     sub read_new_tty {
130
131
132         open(TTY, "<$TTY_PATH");
133         print "Waiting for message: ";
134         ReadMode "raw";
135         my $string;
136
137         while ( my $key = ReadKey 0, *TTY ){
138             $string .= $key;
139         }
140         ReadMode "normal";
141         close(TTY);
142             my @content=split(":",$string);        #splits message
143             $SHIP_ID="$content[0]";
144             $VM_MSG="$content[1]";
145         print "Got string: '$string'\n";
146         return $string;
147     }
```

8.4 return_answer.pl

```
#!/usr/bin/perl

use strict "vars";
use strict "refs";
use Getopt::Std;
use threads;
use Time::HiRes;

use Time::HiRes qw(usleep);use List::MoreUtils qw(uniq);

my $TTY_PATH;       #pts path to VM
my $READ_QUEUE="compute08"; #Queue for incoming traffic
my $WRITE_QUEUE; #Queue for outgoing traffic
my $CONN; #Global connectection to rabbitMQ
my $CHAN; #Channel for rabbitMQ connection
my $ANYEVENT;
my $MSG_SENT = 0;
my $VM_UUID; #This is the UUID for the vm to recieve traffic. This is intended for collecting the
correct tty channel
my $VM_MSG; #Message which is forwarded to vm
my $VM_RESPONSE;        #Response which is sent from VM based on the previous message it
recieved
my $SHIP_ID; #Current shipment ID for message
my $SIM = 1;

print "Handle message called for '$ARGV[0]\n";

my $body=$ARGV[0];

construct($body);
print "Tread constructed, sleeping\n";
print "Sleeping 0.62 seconds\n";
usleep(61852);
write_to_queue("e606cfc4-3900-4cd9-b429-d47dae526179:Fake Return");
print "Finished writing message to queue\n";

if(!$SIM){
    print "Using tty: $TTY_PATH\n";
```

```
    print "This is the correct return queue: $WRITE_QUEUE\n";
    print "This is the correct return shipment ID: $SHIP_ID\n";
    print "this is vms_msg:$VM_MSG\n";
    write_tty($VM_MSG);
}
print "joining threads\n";



sub construct{
        my $body = $_[0]; #message from RabbitMQ
        my @content=split(":",$body);        #splits message
        $WRITE_QUEUE=$content[1];        #Stores correct return queue
        $VM_UUID=$content[0];        #Stores UUID for forwarding message through serial interface
        $SHIP_ID=$content[2];
        $VM_MSG="$SHIP_ID:$content[3]";
}

sub write_to_queue{

    my $msg = $_[0];
    print "Attempting to return message: '$msg'\n";
    use Net::RabbitFoot;
    my $CONN = Net::RabbitFoot->new()->load_xml_spec()->connect(
        host => '10.10.10.51',
        port => 5672,
        user => '*',
        pass => '*',
        vhost => '/',
    );


    my $CHAN = $CONN->open_channel();

    $CHAN->declare_queue(
        queue => $WRITE_QUEUE,
        durable => 1,
    );


    $CHAN->publish(
        exchange => '',
        routing_key => $WRITE_QUEUE,
            body => $msg,
        ) or warn("Could not print message...\n");
}

sub listen_tty{
    print "listen tty called\n";
    my $tobject = threads->self;
    my $tid = $tobject->tid;
    print "Listen is tid $tid\n";
    if($SIM){
        print "Sleeping 0.62 seconds\n";
        usleep(61852);
        write_to_queue("e606cfc4-3900-4cd9-b429-d47dae526179:Fake Return");
        print "Finished writing message to queue\n";
    }
}
```